

DEBUGGING WITH DOMAIN-SPECIFIC EVENTS VIA MACROS

by
Xiangqi Li

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

School of Computing
The University of Utah
December 2017

Copyright © Xiangqi Li 2017

All Rights Reserved

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of Xiangqi Li
has been approved by the following supervisory committee members:

<u>Matthew Flatt</u>	, Chair	<u>10/05/2017</u> Date Approved
<u>Eric Eide</u>	, Member	<u>10/05/2017</u> Date Approved
<u>John Regehr</u>	, Member	<u>10/02/2017</u> Date Approved
<u>Matthew Might</u>	, Member	<u>10/20/2017</u> Date Approved
<u>Sean McDirmid</u>	, Member	<u>10/11/2017</u> Date Approved

and by Ross Whitaker, Chair/Dean of
the Department/College/School of Computing

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Domain-specific languages (DSLs) are increasingly popular, and there are a variety of ways to create a DSL. A DSL designer might write an interpreter from scratch, compile the DSL to another language, express DSL concepts using only the existing forms of an existing language, or implement DSL constructs using a language's extension capabilities, including macros. While extensible languages can offer the easiest opportunity for creating a DSL that takes advantage of the language's existing infrastructure, existing tools for debugging fail to adequately adapt the debugging experience to a given domain.

This dissertation addresses the problem of debugging DSLs defined with macros and describes an event-oriented approach that works well with a macro-expansion view of language implementation. It pairs the mapping of DSL terms to host terms with an event mapping to convert primitive events back to domain-specific concepts. Domain-specific events can be further inspected or manipulated to construct domain-specific debuggers.

This dissertation presents a core model of evaluation and events and also presents a language design—analogue to pattern-based notations for macros, but in the other direction—for describing how events in a DSL's expansion are mapped to events at the DSL's level. The domain-specific events can enable useful, domain-specific debuggers, and the dissertation introduces a design for a debugging framework to help with debugger construction. To validate the design of the debugging framework, a debugging framework, Ripple, is implemented, and this dissertation demonstrates that with a modest amount of work, Ripple can support building domain-specific debuggers.

CONTENTS

ABSTRACT	iii
LIST OF FIGURES	vi
ACKNOWLEDGMENTS	viii
CHAPTERS	
1. INTRODUCTION	1
1.1 Thesis Statement	3
1.2 Thesis Outline	3
2. EXISTING DEBUGGING TECHNIQUES	5
2.1 Stepping-Based Debugging	5
2.2 Scriptable Debugging	5
2.3 Trace-Oriented Debugging	6
2.4 Event-Based Debugging	7
2.5 Aspect-Oriented Debugging	7
3. MEDIC: METAPROGRAMMING AND TRACE-ORIENTED DEBUGGING	9
3.1 A Trace-Oriented Metaprogramming Language	9
3.2 Trace Debugging	12
3.2.1 Log Tracing	12
3.2.2 Graph Tracing	14
3.2.3 Aggregate Tracing	16
3.2.4 Timeline Tracing	20
3.3 Implementation	23
3.3.1 Interpretation of Medic Programs	23
3.3.2 Weaving Debugging Code	23
3.3.3 Generation and Presentation of Traces	28
4. DEBUGGING WITH DOMAIN-SPECIFIC EVENTS	30
4.1 Motivation	30
4.2 Implementing DSLs with Macros	33
4.3 Core Events	38
4.3.1 Completeness of Model	47
4.4 Mapping Events	52
4.4.1 Declaring Events	52
4.4.2 Examples and Kinds of Event Mappings	54
4.4.3 Environment Information in Events	57
4.4.4 Continuation Information in Events	59

4.5	Run-Time Event Generation	59
4.5.1	Event Dependency Construction	60
4.5.2	DSL Event Generation	62
5.	DEBUGGING FRAMEWORK	65
5.1	Front End	65
5.1.1	Elementary Mode	65
5.1.2	Motivation for Other Possible Modes	68
5.1.3	Intermediate Mode	69
5.1.4	Advanced Mode	70
5.1.5	Front-End Construction	70
6.	APPLICATIONS	71
6.1	Scratchy Debugger	71
6.1.1	Finding Scratchy Bugs in the Elementary Mode	75
6.1.2	Finding Scratchy Bugs in the Intermediate Mode	75
6.2	POP-PL Debugger	81
6.2.1	Finding POP-PL Bugs in the Elementary Mode	84
6.2.2	Finding POP-PL Bugs in the Advanced Mode	86
6.3	Medic Debugger	89
6.3.1	Demonstration of Finding Bugs	91
7.	EVALUATION	92
7.1	Support for Domain Customizations	92
7.2	Support for Effective Debuggers	93
7.3	Ease of Debugger Construction	96
7.4	Performance	98
8.	OUTLOOK	100
8.1	Composition of DSLs	100
8.1.1	Debugging Coexisting DSLs	101
8.2	Improving the Debugging Experience	105
9.	RELATED WORK	108
9.1	Event-Based Debugging for GPLs	108
9.2	Debugging Support on Language Workbenches	109
9.3	Other Debugging Support for DSLs	110
	REFERENCES	112

LIST OF FIGURES

3.1	Medic grammar	10
3.2	Medic location grammar	10
3.3	Showing the behavior of data	13
3.4	Showing the layer of interest	15
3.5	The buggy remove implementation	17
3.6	The log view of a doubly linked list	18
3.7	Graph tracing	18
3.8	The graph view of a doubly linked list	19
3.9	Timeline tracing	21
3.10	Timeline traces for timeline form	21
3.11	Time view	22
3.12	Debugging tables	24
3.13	Retrieval and transformation of syntax objects	26
3.14	Process of weaving module-level code	26
3.15	Process of weaving expression-level code	27
4.1	POP-PL debugger	32
4.2	A comparison of the traditional approach (top) and my approach (bottom) to DSL debugger construction	32
4.3	Macro expansion	34
4.4	An overview of layered DSL implementation and domain-specific event support	38
4.5	The CESKT machine	40
4.6	Grammars for the extended CESK machine	41
4.7	An event framework for domain-specific event creation and generation	61
4.8	Event-related structure representation	61
5.1	Debugger construction process	67
6.1	Scratchy debugger	72
6.2	Scratchy debugger interface in an elementary mode	72
6.3	Scratchy debugger interface in an intermediate mode	74

6.4	Event assertion	74
6.5	A buggy Scratchy program	76
6.6	Scratchy debugger interface after first step	77
6.7	Scratchy debugger interface after second step	77
6.8	Scratchy example	78
6.9	Enabling stepping in the intermediate mode	80
6.10	Stepping to the bottom wall statement	80
6.11	Stepping to the top wall statement	81
6.12	POP-PL debugger interface in an elementary mode	82
6.13	POP-PL debugger interface in an advanced mode	83
6.14	Debugging scenario in the elementary mode	85
6.15	Debugging scenario in the advanced mode	88
6.16	Exploring with timelines in the advanced mode	89
6.17	Medic debugger	90
7.1	Ripple support for domain customizations	92
7.2	Size of debugger implementation	98
7.3	Time overhead of debuggers	99
7.4	Memory usage	99
8.1	Gate debugger	103
8.2	Device debugger	103
8.3	Circuit debugger	104
8.4	Debugger interface for composition of DSLs	106

ACKNOWLEDGMENTS

A dissertation is not conceived and born in isolation. A work as large and complicated as a research project and dissertation requires the help and support of many people and has many parents. People, some fixtures in your life and others passing through, offer moral support, set helpful examples, or become sources of inspiration. I would like to take the opportunity to pay tribute to some of the people who helped or inspired me over the course of my life and my graduate work.

First, my parents, my grandparents, and my other family relatives provided a loving home for me and always encouraged me to pursue academic studies and to become a productive person. My father's determination in study and problem-solving had a great influence on me and made me believe that nearly anything can be done with hard work and persistence. Since I was a child, my grandmother has helped to care for me. She was always able to understand me, and she provided a lot of emotional support for me as I undertook my academic journey.

The person to whom I am most grateful is my advisor, Matthew Flatt. The academic environment in America differs from the environment in China in many ways, and when I started my studies, I needed to transition into a foreign academic environment and into an academic program so different from the program in my college education. I suffered through a lot of struggles and doubts, but my advisor was always the strongest of supports for me and guided me through the intricacies of graduate study. He introduced me to the academic world and taught me how to conduct research and be a researcher. His attitude to work, his expertise, and his passion left an indelible impression on me.

In the process of studying for the doctorate, many faculty and friends have had a positive effect on me. I would like to recognize several people in particular. Professor Robert Findler and Professor Matthias Felleisen gave me invaluable reassurance and academic guidance at some of the lowest points in my study. My mentor, Sean McDirmid, at Microsoft Research, gave me another perspective on research. That the members of my committee gave me important assistance goes without saying, and I would like to thank the committee members not previously mentioned, Eric

Eide, Matthew Might, and John Regehr, for all of their patience and help. Of course, graduate study would not be the same without the company of peers, my friends, Kevin Tew, Jon Rafkind, Danny Yoo, Yang Chen, Jubi Taneja, and William Hatch, with whom I shared ups and downs, the thrills and anxieties of graduate work.

Finally, I would like to thank my husband who lovingly and patiently endured the worst of my fears and pushed me to believe in myself and to continue to strive when I doubted that I had the talent or the imagination to succeed. No words of acknowledgment can make up for all of the tension. I really can't thank him enough.

CHAPTER 1

INTRODUCTION

Domain-Specific Languages (DSLs) have recently attracted increasing attention in the area of programming languages. They are designed to solve problems in a particular domain with expressive, domain-specific notations and abstractions (van Deursen et al. 2000) and to achieve ease of use and gains in productivity (Mernik et al. 2005; Ward 1994). Many tools exist to facilitate developing and using domain-specific languages such as parser generators (Visser 1997), integrated development environments (IDEs) (Charles et al. 2009; JetBrains 2004; Kats and Visser 2010), and program transformations (Bravenboer et al. 2008). With current DSL-construction tools, however, debugging support for DSLs remains unsatisfactory.

Currently, there are two popular kinds of debuggers. The first kind is a generic, general-purpose language (GPL) debugger, like GDB (Stallman et al. 2002), capable of debugging multiple GPLs. The traditional GPL debugger relies on a stepping-based approach, which supports controlling the execution of a program by setting breakpoints and stepping. Due to the abstraction gap between a GPL and a DSL, using the traditional GPL debugger to debug DSLs is not effective. The second kind is a specialized DSL debugger, which provides a most useful debugging experience by tailoring the debugger to DSL needs. A specialized DSL debugger is effective, but designing and developing a specialized debugger for each DSL tends to be expensive and unrealistic.

To reduce the cost of building a DSL debugger, there has been a movement toward DSL debugging frameworks (Chis et al. 2014; Henriques et al. 2005; Lindeman et al. 2011; Van Den Brand et al. 2005; Wu et al. 2008). However, these DSL debugging frameworks exhibit limitations. Sometimes, the problem is that debugging operations are limited to traditional debugging techniques for imperative languages, such as setting breakpoints and stepping (Henriques et al. 2005; Van Den Brand et al. 2005; Wu et al. 2008). Even in an event-based debugging framework, often the events are insufficiently general (Chis et al. 2014; Lindeman et al. 2011) and expose too much the constructs of the GPL that hosts the DSL implementation.

In the realm of debugging for GPLs, a programmer is usually presented with an interface to step through imperative statements and set breakpoints. A more general model, which is supported by many debuggers, views the activity of a program as a generator of events that can be inspected and to some degree controlled (Bates 1995; Marceau et al. 2006; Olsson et al. 1990). An event-based view enables reuse and extension of a debugger, and I take this view as one of my starting points. In the same way that GPLs can host DSL implementations, an environment that provides GPL debugging events can host DSL debugging events. This approach adapts well to many different kinds of problems, including domain-specific problems where the evaluation rules might not follow a conventional imperative flow. A gap remains, however, for mapping GPL events back to a DSL in a precise, reusable, and extensible way.

I combine an event-oriented view of DSL debugging with a view of DSL construction based on macros, as they are commonly implemented in Lisp environments and especially in Racket (Felleisen et al. 2015; Tobin-Hochstadt et al. 2011). Macros provide high-level support for converting the constructs of a DSL into lower-level constructs of a host language. Macros compose well, so that multiple DSLs can coexist in a larger application along with the host language. Macros also naturally enable towers of languages (Ward 1994), where terms in a source language are expanded to successively simpler languages, while each intermediate point serves as a well-defined and reusable language in its own right.

An event-oriented view of debugging meshes well with a macro-expansion view of language implementation. Macros specify the run-time semantics of a DSL through elaboration into lower-level constructs. Meanwhile, debugging events from the lower-level language can be filtered, combined, and transformed to describe debugging events in terms of the DSL. In the same way that static elements of a language can be associated with macros to implement, say, a type system (Chang et al. 2017), a protocol for debugging events can be integrated with macro transformations. As a result, the programmer gets support for DSL debugging on par with support for DSL type systems and run-time evaluation. I further design and develop a debugging framework that incorporates a suite of visualization and interaction tools to allow programmers to view and interact with events to debug programs, where the visualizations and interactions can be tailored to the DSL as needed.

1.1 Thesis Statement

To assist creation and integration of DSLs, language workbenches (Efftinge and Volter 2006; JetBrains 2004; Kats and Visser 2010; Krahn et al. 2008) are recent platforms incorporating tools for most aspects of DSLs, which include means for language creation, interpreter or compiler support, and IDE services such as syntax checking and syntax completion. DSLs are still recognizably programming languages, and a good debugging support is indispensable to using any programming language. However, the existing support for debugging DSLs on most language workbenches is limited.

This dissertation presents an approach to debugging DSLs, executable DSLs, on the Racket language workbench that implements DSLs via macros. The thesis statement is: **A workbench for building domain-specific programming languages can offer not only tools for defining the static and run-time semantics of the language but also a debugging framework to ease the development of effective, domain-specific debuggers.**

I build on the idea of debugging as the inspection and manipulation of domain-specific events that are generated by a program's execution. Whereas macros define a DSL's semantics by rewriting source terms to more primitive constructs, I use *event mappings* in macro transformations to convert primitive events back to domain-specific concepts. Domain-specific events are then suitable for presenting to a user or wiring into a domain-specific visualization. To reduce the cost of debugger construction, I designed and developed a debugging framework, Ripple. Ripple partitions the debugger support into a back end and a front end where the back end facilitates debugging information collection and where the front end facilitates interface construction.

1.2 Thesis Outline

Chapter 2 gives an overview of existing debugging techniques that dominate the field of debugging, which are stepping-based debugging, scriptable debugging, trace-oriented debugging, event-based debugging, and aspect-oriented debugging.

Chapter 3 is an initial exploration of instrumentation without using events in its debugging model. I present a new debugging and program-exploration tool, Medic, that solves the problems of trace debugging. The development of traces helped to shape my later design of a debugging framework. After initial development, Medic evolved to use events in its implementation and turned out to be an example of a DSL for the rest of my dissertation work.

Chapter 4 lays a foundation for evaluation and describes domain-specific events that support the back end of my debugging framework in Chapter 5. I present a core programming-language model that supports debugging events, where the events are sufficient to fully reconstruct the state of an evaluation. I then describe the constructs that DSL implementers use to map events from one language level (either the core language or a derived language) to a new language level; those constructs are ultimately implemented in terms of the core language’s event-reporting mechanism, but with conversion layers that aggregate and transform core events into domain-specific events.

Chapter 5 presents the design of my debugging framework. The debugging framework incorporates a suite of tools to support building effective, domain-specific debuggers with low cost.

Chapter 6 presents applications of the debugging framework. To help validate the design of my debugging framework, I have implemented debugging support for three DSLs, and I report my experience in finding bugs with these debuggers.

Chapter 7 is an evaluation of the usefulness of the debugging framework. I have tested the framework on three DSLs, and I demonstrate that with a modest amount of work, my debugging framework can support constructing effective, domain-specific debuggers.

Chapter 8 presents a perspective on the debugging framework and discusses the issues of composition of DSLs and the future work.

Chapter 9 describes the related work about event-based debugging for GPLs and debugging support for DSLs.

CHAPTER 2

EXISTING DEBUGGING TECHNIQUES

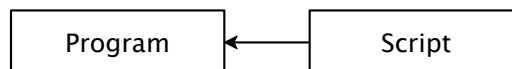
There are many debugging techniques that are proposed in the area of debugging, and this chapter gives a background of five popular debugging techniques: stepping-based debugging, scriptable debugging, trace-oriented debugging, event-based debugging, and aspect-oriented debugging.

2.1 Stepping-Based Debugging

A stepping-based approach relies on the concepts of setting breakpoints and stepping. The debugger provided in the Eclipse IDE (Eclipse 2017) represents a common example of a stepping-based debugger. The program execution can be controlled by breakpoints where a *breakpoint* is a location specified in the source program to enable suspension of execution. A stepping-based debugger also supports stepping-related operations including *step*, *step over*, *step into*, and *step out*. The step operation supports stepping through the execution of a program in a line-by-line fashion or a statement-by-statement fashion. The step over operation supports executing a procedure in one step. The step into operation jumps into the first statement of a procedure, and the step out operation finishes executing the remaining code in a procedure if the execution steps into a procedure. When the execution is paused, the debugger shows the current program states in terms of variables and stack frames.

2.2 Scriptable Debugging

A scriptable debugger treats debugging as a programmable and repeatable activity where the debugger provides a debugging language to assist debugging. For users, the debugging process involves using the debugging language to write a script to describe debugging operations on a program:



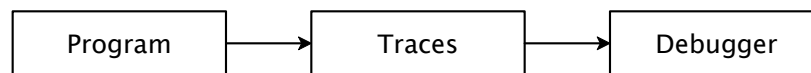
Examples of scriptable debuggers are MzTake (Marceau et al. 2006), Dalek (Olsson et al. 1990),

Duel (Golan and Hanson 1993), and RAIDE (Johnson 1977). A scriptable debugger usually provide a debugging language, which differs in syntax from the language used in other scriptable debuggers, and the debugging language is able to express operations that are common to debugging such as inspecting variable values and controlling program execution.

Because scriptable debuggers incorporate an expressive language, using a scriptable debugger offers exceptional debugging power by enabling the scripting of debugging tasks. Debugging scripts have the benefit of eliminating repetitive debugging actions and promoting modular organization and reuse of debugging code. However, because of the dependence on a debugging language, the traditional, graphical operations are replaced by textual scripts where learning the debugging language is a prerequisite to begin any debugging activities. Most debugging languages are system-dependent, and users need to learn a new language when switching to a different debugging system.

2.3 Trace-Oriented Debugging

A trace-oriented debugger instruments the program to produce *traces* at run time, and traces are directed to the debugger for inspection and manipulation:



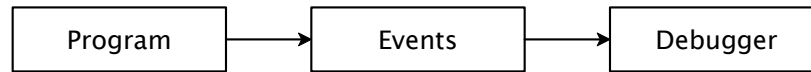
Trace-oriented debugging includes `print` debugging, where the traces are generated by instrumentation of `print`-like statements, and omniscient debugging, where the traces are generated internally by the system. The `print` debugging method has little cognitive load because the method only requires the insertion of `print` statements into the source program, which needs no debugger support and specific debugging knowledge. Omniscient debugging supports back-in-time debugging where the execution histories are all recorded and where users can examine the past execution to find problems. Examples of omniscient debuggers include ODB (Lewis 2003) and TOD (Pothier et al. 2007).

Trace-oriented debugging enjoys the benefit of never losing any data of interest during program execution, but different approaches suffer different problems. Though simple, `print` debugging bears several limitations. First, the source program is tangled with `print` statements, which obscures the original code during debugging and requires removing the `print` statements after debugging. Second, the trace output is usually textual and linear, which is difficult to analyze for voluminous traces. Omniscient debugging saves every execution state, which contains information

about variables and stack frames, suffering a bigger problem of space consumption.

2.4 Event-Based Debugging

Event-based debugging views the activity of a program as a generator of *events* that can be inspected and to some degree controlled by debuggers:



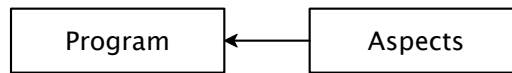
An event represents the occurrence of an activity during the execution of a program, and event-based debugging is similar to trace-oriented debugging if the traces are based on events. Examples of event-based debuggers include Dalek, UFO (Auguston et al. 2003), EBBA (Bates 1995), and MzTake. A debugger can employ more than one category of debugging techniques. For example, Dalek as well as MzTake are both scriptable and event-based.

Event-based debuggers generally use events as information units where debugging information is encapsulated in event attributes, but different debuggers use events differently to support various models of debugging. For example, UFO treats debugging as a computation over an event trace (execution histories), which can be performed either at run time or after execution, while EBBA treats debugging as an activity of matching the actual program behavior to the expected program behavior expressed by events. Meanwhile, the model of events and creation of new events varies among systems. UFO views events as an activity that takes place in a time interval, and other systems model events as points of time. Dalek and EBBA support creating high-level events that are defined in terms of other events.

2.5 Aspect-Oriented Debugging

Aspect-oriented debugging originates from aspect-oriented programming (AOP) (Kiczales et al. 1997). AOP tries to modularize crosscutting concerns that are usually scattered around the code and are difficult to develop and maintain. AOP typically involves concepts of *join points*, *pointcuts*, *advice*, and *aspects*. A join point represents a location in the execution of a program, and a pointcut provides a way to specify a collection of join points. Advice describes any concerns that will work on pointcuts to influence the original code, which is usually written in a method-like construct. With components of join points, pointcuts, and advice, aspects are units of modularization that express the implementation of crosscutting concerns.

When the crosscutting concerns are related to debugging, AOP becomes aspect-oriented debugging, and the debugging process is:



Similar to scriptable debugging, the original source program remains intact during debugging, and aspects, which encapsulate debugging operations, work on the source program to automate debugging tasks. Most AOP models (Dutchyn 2012; Kiczales et al. 2001; Spinczyk et al. 2002) provide limited join-point models and limit access to values in advice, which affect the effectiveness of debugging. There are aspect-oriented systems such as Bugdel (Usui and Chiba 2005) that try to tackle the limitations of AOP and aim to provide a useful debugging experience. Overall, aspect-oriented debugging enjoys the benefits of AOP, which include preserving the modularity and encapsulation of the target language and supporting the organization and reuse of debugging code, but has some limitations in supporting a full-featured debugging.

CHAPTER 3

MEDIC: METAPROGRAMMING AND TRACE-ORIENTED DEBUGGING

Even though modern programmers enjoy a wealth of high-level and graphical tools for understanding and debugging programs, programmers often resort to the simple and the time-honored technique of inserting `print` statements into programs to reveal progress and to expose intermediate values. This technique is called *trace debugging* or *printf debugging*. Compared to other debugging techniques, trace debugging offers many benefits such as lightweight `print` statements and the resulting convenience in exposing program states. However, traditional trace debugging also has several drawbacks, including the need to modify the source program and the need for additional tools when trace output becomes too voluminous.

This chapter is my exploration of debugging support. I started with the debugging support for GPLs, and I designed and implemented Medic to improve trace debugging. Medic, a new debugging and program-exploration tool for Racket, augments the traditional examination of control and state with output processing, metaprogramming, and visualization features. Medic allows programmers to leverage the benefits of trace debugging while addressing many of its drawbacks. The work of Medic used ideas of metaprogramming, data inspection, and data visualizations, which turned out to be useful for debugging support for DSLs. A variant of this chapter was originally published as *Medic: Metaprogramming and Trace-Oriented Debugging* (Li and Flatt 2015) in the Workshop on Future Programming, <http://dx.doi.org/10.1145/2846656.2846658>.

3.1 A Trace-Oriented Metaprogramming Language

A Medic programmer uses a metaprogramming language to inject “printing” forms into the program to be debugged. Figure 3.1 shows the overall grammar of a metaprogram; the grammar references the *mod-scope* definition in Figure 3.2, and we begin our explanation with *mod-scope*.

Medic’s metaprogramming facilities start with the ability to describe the placement of debugging instructions. Programmers must specify the *locations* in the source program where the debug-

```

top-level-form ::= layer-def ...
  layer-def ::= (layer layer-id layer-form ...)
              | (layer layer-id #:enable flag layer-form ...)
layer-form ::= port-form
            | debug-def
            | mod-scope
port-form ::= (export id ...)
            | (import layer-id ...)
debug-def ::= (define-source debug-src-id source-expr)
            | (define-source (debug-src-id arg-id ...)
                          source-expr ...)
            | (define-match debug-id match-form)
            | (define-match (debug-id arg-id ...)
                          match-form ...)
behavior-form ::= (with-behavior f template)
               | (with-behavior f template #:renamed ret id)
match-ref-form ::= debug-id
                 | (debug-id match-arg ...)
src-ref-form ::= debug-src-id
              | (debug-src-id src-arg ...)

flag ::= boolean
template ::= a @ form describing function behavior
layer-id, id, f ::= an identifier
debug-src-id, debug-id, arg-id ::= an identifier

```

Figure 3.1. Medic grammar

```

mod-scope ::= (in #:module module-name match-form ...)
match-form ::= behavior-form
            | fun-scope
            | match-ref-form
            | insert-form
fun-scope ::= [each-function insert-form ...]
           | [(f ...) insert-form ...]
insert-form ::= border-form
             | at-form
border-form ::= [on-entry source-expr ...]
             | [on-exit source-expr ...]
at-form ::= [at location-form before-form after-form
            border-form ...]
before-form ::= #:before location-form | ()
after-form ::= #:after location-form | ()
location-form ::= target-language-expression
               | expression-pattern

module-name ::= any
source-expr ::= src-ref-form
              | target-language-expression
target-language-expression ::= a legal expression in the target language
expression-pattern ::= an expression pattern

```

Figure 3.2. Medic location grammar

ging instructions are added. Medic provides three categories of *scope* specification of locations:

- **Module-level.** Racket organizes programs into modules, where each module has its own global namespace and resides in its own file, so all location descriptions start by identifying the relevant module. The `(in #:module module-name match-form ...)` form confines the scope of *match-forms* to be within the module named *module-name*.
- **Function-level.** Within a module, a function-level scope limits the functions where insertion specifications apply. Medic supports two kinds of function-level scopes: `each-function`, which matches every function, and `(f ...)`, which specifies one or more functions by name.
- **Expression-level.** The *at-form* pattern locates a specific expression in the source program that matches a *location-form*. The *location-form* can be a complete expression or an expression pattern where an underscore in the pattern matches any expression. For example, `(+ _ 1)` matches `(+ x 1)`, `(+ (f 3) 1)`, and `(+ (if (zero? x) 4 5) 1)`. Patterns specified after `#:before` and `#:after` keywords can further constrain the matching of target expressions.

After a location is identified in the source program, debugging code can be inserted either on entry or on exit using the `[on-entry source-expr ...]` or `[on-exit source-expr ...]` form. While some AOP systems constrain inserted code so that it preserves modularity and encapsulation properties of the original program, Medic imposes no constraints, as is appropriate for debugging purposes. Medic thus allows an inserted form to access any identifier that is visible in the source program at the insertion point.

Medic also provides a means of modularity and abstraction. A Medic program consists of different *layers*, where a layer modularizes debugging code for a specific functionality and groups traces produced by the `print` form. A layer is declared by `(layer layer-id layer-form ...)` with an optional `#:enable` specification that can enable or disable a layer's output.

Within a layer, the *debug-def* form enables abstraction and parameterization of a fragment of code. The code fragment can be either run-time code to inject into the source, as indicated by the `define-source` form, or it can be a fragment of metaprogramming specification, as indicated by the `define-match` form. These definitions are referenced using *src-ref-form* and *match-ref-form*, which refer to the corresponding fragment of code named by a *debug-src-id* or *debug-id* in a suitable context. Definitions within a layer are exported to other layers and imported from other layers using

(`export id ...`) and (`import layer-id ...`), respectively. A disabled layer’s definitions remain available for import by other layers.

Besides allowing access to elements of a source program in debugging instrumentation, Medic reflects additional information for use in certain debugging forms. The `function-name` variable is bound to the enclosing function or a function being called. For example, instead of tediously annotating each function `f`, `g`, etc., with (`print "f function entered"`), (`print "g function entered"`), etc., Medic allows writing

```
[each-function [on-entry @log{@function-name function entered}]]
```

The string-template notation using `@` is an alternate representation of an S-expression, and is the same as used for Scribble (Barzilay 2009; Flatt et al. 2009). Through (`with-behavior f template`), a programmer can define the logging behavior, represented by *template*, of the *f* function. Inside *template*, `arg` and `ret` are allowed to escape to access a function’s argument and return value.

3.2 Trace Debugging

Medic supports four kinds of tracing (i.e., variants of `print`) with associated visualizations, each of which is useful for different debugging tasks. All traces are directed to an interactive graphical user interface, the *trace browser*. The trace browser consists of four panes that correspond to the four kinds of traces: a Log pane, a Graph pane, an Aggregate pane, and a Timeline pane.

3.2.1 Log Tracing

Log tracing with Medic’s `log` form is similar to traditional trace debugging with `print`, but Medic’s logging facilities simplify the construction and browsing of printed output. The `log` form not only produces traces sequentially in execution order, but it also augments traces with useful context information. For example, suppose that the value of `x` is 3. The (`log x`) expression produces a “`x = 3`” trace entry in the Log pane of the trace browser. Unlike the traditional `print` statement, which merely prints `x`’s value, `log` produces extra context information about the value: the name of the variable under inspection. This automatic addition of context by `log` relieves the programmer of tedious string-templating work for simple logging output.

The `log` form recognizes function calls as well as variable references, and it cooperates with Medic statements that adjust the format of output for function calls. A programmer can control the way that log output is written for `f` calls by using (`with-behavior f template`). This centralized

control provides an alternative to duplicating template constructions at each logging site. Furthermore, the `with-behavior` form provides access to the result of the function calls, as well as the arguments, which allows the `log` form to more completely expose the behavior of the function. For example, suppose a `f.rkt` program. One can log the behavior of calls to the `f` function by writing a Medic program in `f-medic.rkt` as shown in Figure 3.3. The `with-behavior` clause in `f-medic.rkt` changes the output produced by `log` whenever a function call to `f` is logged. Instead of printing just the result of calling `f`, `log` displays the customized behavior of `f`—“`f: @x squared plus @y squared is @ret`”—with `@x`, `@y`, and `@ret` replaced by arguments’ values and the return value of `f` function call. After starting a debugging session by running `f-medic.rkt` and `f.rkt`, the log entries are the following (traces showing the behavior of data are highlighted in blue):

```
f: 3 squared plus 4 squared is 25
f: 4 squared plus 5 squared is 41
```

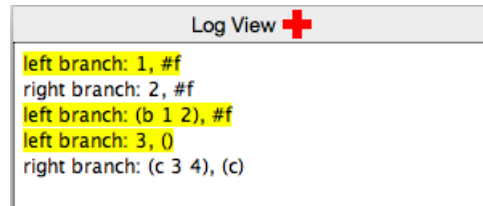
One of the most difficult aspects of trace debugging is determining the right amount of data to log. Logging too little data defeats the point, but logging too much data makes the interesting information difficult to find, and the right amount of logging is not always clear from the start. Medic’s *layers* help programmers organize output so that layers of output can be selected interactively, which helps balance the needs of showing enough information and limiting the amount of information.

```
f.rkt
(define (f x y)
  (+ (sqr x) (sqr y)))

f-medic.rkt
(layer layer1
  (in #:module "f.rkt"
    (with-behavior f
      @f: @x squared plus @y squared is @ret)
    [on-exit (log (f 3 4))
            (log (f 4 5))]))
```

Figure 3.3. Showing the behavior of data

Figure 3.4 shows an example of defining layers for traces. To see the traces produced by `(log "left branch: ~a, ~a" (cadr t) left-p)`, the programmer can click the *Log View* button, which opens a layer-view window listing the layers of traces: `left-path` and `right-path`. After selecting the `left-path` layer, the Log pane updates the display of traces immediately, highlighting the traces that belong to the selected layer:



Seeing traces in layers enables a programmer to make better comparisons of relevant trace elements among all mixed traces, while the execution order of traces is preserved.

3.2.2 Graph Tracing

Traces produced by `log` are linear and text-based. They print primitive values in a typical form, and by preserving the execution order of traces, they enable analysis of the evolution of a value in a program. Textual traces, however, provide a poor view of certain relationships among trace elements that could become immediately apparent in a graph view. With conventional logging tools, converting textual output to a graph view requires additional tools, careful formatting of output to fit the tools' input formats, and isolation of that output from other debugging output.

Medic directly supports the construction of *graph* output to help programmers see the otherwise hidden relationships among values. To generate a graph, instead of using `log`, the programmer uses the `node` and `edge` forms. The `node` form takes a value, an optional label string, and an optional color for the node. The `edge` form takes two values, an optional edge label, optional color, and optional labels for the two ends of the edge. Naturally, `node` creates a node in the visualization, and `edge` creates an edge between nodes (either declared previously or implicitly created by `edge`). Medic also supports `remove-node` and `remove-edge` forms to delete the graph trace's nodes and edges. To create a simple and aesthetically pleasing visualization, Medic uses force-directed algorithms to lay out the output (Eades 1984; Fruchterman and Reingold 1991).

To illustrate Medic's graph tracing facilities, suppose that we have a correct implementation of a doubly linked list with support for common accessing, inserting, and removing operations. The

find-path.rkt

```

(define (find-path t name)
  (cond
    [(string? t) (if (equal? t name) '() #f)]
    [else
     (let ([left-p (find-path (cadr t) name)])
       (if left-p
           (cons (car t) left-p)
           (let ([right-p
                  (find-path (caddr t) name)])
             (if right-p
                 (cons (car t) right-p)
                 #f)))))]))

(find-path '("a" ("b" "1" "2") ("c" "3" "4")) "3")

```

find-path-medic.rkt

```

(layer left-path
  (in #:module "find-path.rkt"
    [at (if left-p _ _)
      [on-entry
        (log "left branch: ~a, ~a"
              (cadr t) left-p)]]))

(layer right-path
  (in #:module "find-path.rkt"
    [at (if right-p _ _)
      [on-entry
        (log "right branch: ~a, ~a"
              (caddr t) right-p)]]))

```

Figure 3.4. Showing the layer of interest

`remove` method takes an argument, `i`, and removes the `i`th element from the list starting from index 0. We can create a bug in the `remove` implementation by commenting out the line of code that updates the previous link of a node, `temp-next`, to point to the node, `temp-prev`, when the node, `temp`, is to be deleted, as shown in Figure 3.5.

To test the broken library, we add ten numbers from 0 to 9 to the doubly linked list `dlist` and then remove five successive elements 3, 4, 5, 6, 7 from the list by calling `(send dlist remove 3)` five times. We can first use `log` to print the elements at each step. From Figure 3.6, we notice we get a faulty list after the removal operation—the final list should be the sequence 0, 1, 2, 8, 9, instead of a sequence of 0, 1, 2, 4, 5. However, the tracing log gives us little insight into the cause of the problem. If we use `edge` to visualize the doubly linked list (see Figure 3.7), we can see the problem instantly. As shown in Figure 3.8, the doubly linked list is broken with a unidirected edge between nodes 2 and 4.

In this example, the test and `edge` declarations were part of the metaprogram. When the library might have its own tests, Medic’s metaprogramming facilities can be used to weave node and edge declarations into the library’s implementation to track down the source of a test failure.

3.2.3 Aggregate Tracing

A programmer can use linear traces with multiple values in each entry to detect a relationship between values and how they change together. A manual inspection of linear output, however, can make those changes difficult to extract from the layout and noise of traces. Medic’s aggregate form presents trace output in a way that makes related output values easier to inspect and compare.

Consider the source program:

```
(define (fact x a)
  (if (zero? x) a (fact (sub1 x) (* x a))))
(fact 3 1)
```

and the Medic program:

```
(layer fact
  (in #:module "fact-iter.rkt"
    [(fact) [on-entry (aggregate x a)]]))
```

In the source program, the `x` and `a` values change together across calls to the function, and inspecting them as a pair can help a programmer understand how they work together. Specifically, using

doubly-linked-list.rkt

```

....
(define/public (remove i)
  (when (or (< i 0) (> i (sub1 size)))
    (error 'remove-invalid-argument))
  (cond
    [(zero? i)
     (define res (get-field datum head))
     (set! head (get-field next head))
     (if head
      (set-field! previous head #f)
      (set! tail #f))
     (set! size (sub1 size))
     res]
    [else
     (cond
      [(= i (sub1 size))
       (define res (get-field datum tail))
       (set! tail (get-field previous tail))
       (set-field! next tail #f)
       (set! size (sub1 size))
       res]
      [else
       (define temp head)
       (for ([j (in-range i)]) (set! temp (get-field next temp)))
       (define res (get-field datum temp))
       (define temp-prev (get-field previous temp))
       (define temp-next (get-field next temp))
       (set-field! next temp-prev temp-next)
       ; (set-field! previous temp-next temp-prev)
       (set! size (sub1 size))
       res]]))
....

```

Figure 3.5. The buggy remove implementation

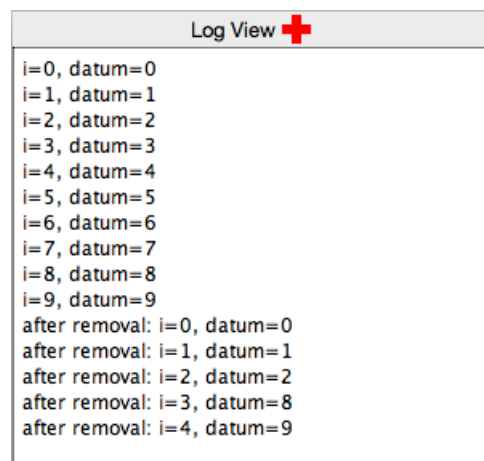


Figure 3.6. The log view of a doubly linked list

```

doubly-linked-list-medic.rkt

(layer dlist
  (in #:module "doubly-linked-list.rkt"
    [on-exit
      (define dlist (new doubly-linked-list%))
      (for ([i (reverse (build-list 10 values))])
        (send dlist add-at 0 i))
      (for ([i (in-range 5)]) (send dlist remove 3))
      (for/fold ([temp (get-field head dlist)]
                ([i (in-range (sub1 (send dlist get-size))]))
                (define next (get-field next temp))
                ; draw an edge: temp -> next
                (edge temp next "" "red"
                  (get-field datum temp)
                  (get-field datum next))
                next)
                (for/fold ([temp (get-field next (get-field head dlist))]
                          ([i (in-range (sub1 (send dlist get-size))]))
                          (define prev (get-field previous temp))
                          ; draw an edge: temp -> prev
                          (edge temp prev "" #f
                            (get-field datum temp)
                            (get-field datum prev))
                          (get-field next temp)))]))

```

Figure 3.7. Graph tracing

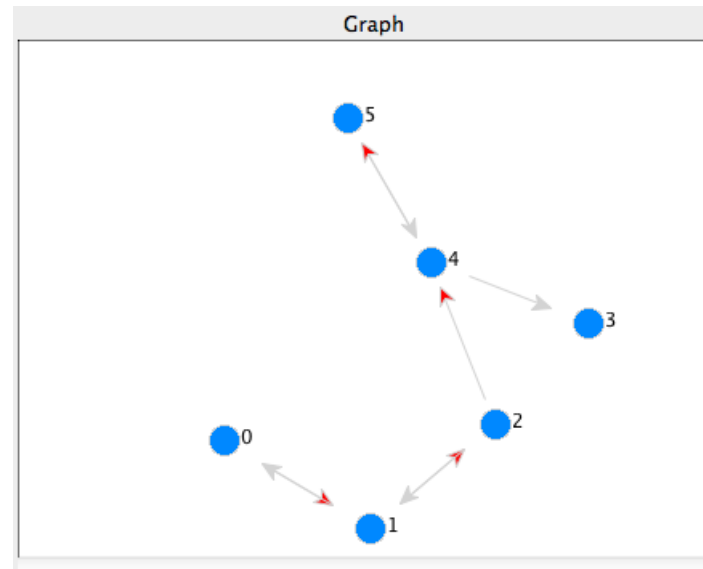

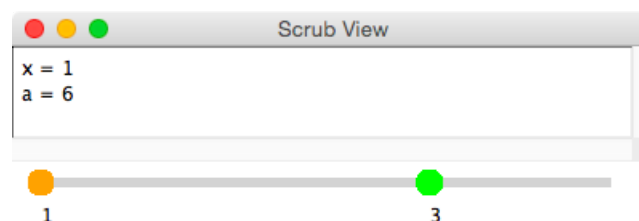


Figure 3.8. The graph view of a doubly linked list

(`aggregate x a`) produces a result that is more organized than a linear trace:

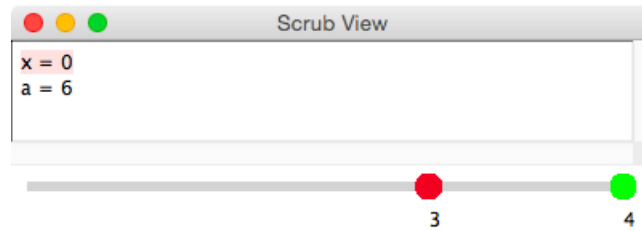
Aggregate				
	x = 3 a = 1	x = 2 a = 3	x = 1 a = 6	x = 0 a = 6

When (`aggregate x a`) is evaluated many times, as in (`fact 1000 1`), the resulting large number of traces must be pruned to expose the values at each step and enable comparison at different steps. Clicking the red button to the left of the `aggregate` trace view opens a scrub-view window, which allows the programmer to inspect the traces step-by-step (currently at step 3):



The scrub view provides two slider handles; the window displays the current step of traces indicated by the second slider handle, but it compares that value to the one selected by the first slider handle. For example, moving the orange slider handle to step 3 and right-clicking on it turns the slider red,

which marks the step for later comparison. Then, moving the green slider handle to step 4 compares the values at step 4 to the values at step 3. The difference between two steps is highlighted in pink:



3.2.4 Timeline Tracing

When traces involve changes over time, programmers need to see the overview of data in a temporal fashion. There are a few possible ways to present traces such as using a slider to “time travel,” using “timeline” views, and using “stroboscopic” views (McDirmid 2013). Inspired by the timeline view of data presented by Victor (2012), which helps programmers understand data with visual context, instead of “peeking through a pinhole,” Medic provides three forms for timeline tracing: `timeline`, `assert`, and `same?`. These three forms all generate traces with a view similar to a timeline view, where each trace element is arranged along the vertical axis, while the changing values of each trace element are displayed along the horizontal axis. By default, the timeline view’s horizontal axis corresponds to an abstract execution time reflecting the order of logged events, but not the delays between events. A clock-based view is available in a separate window.

As a further refinement over aggregate tracing, the timeline view automatically determines a graphical presentation mode for some logged values. For a given element v in a trace, if all occurrences of v are *numbers*, a line plot is rendered on the timeline (where each point is arranged in the square unit according to its numeric value). For *boolean* values, each square unit represents a value with false values colored red and true values colored blue. For other data types, a textual form is displayed.

To illustrate, for the programs shown in Figure 3.9, the left panel of Figure 3.10 presents the resulting timeline view. The timeline slider on the top can step through the timeline traces showing multiple values with the same horizontal coordinates at the same time (see the right panel of Figure 3.10). Clicking the corresponding square unit shows the current value as a tooltip.

A timeline element produced by `(assert pred)` is similar to `pred` as a boolean result, but true values are deemphasized by coloring them in gray, while false elements are highlighted in red.

```

count.rkt
(define (count-length v count)
  (if (null? v)
      count
      (count-length (cdr v) (+ count 1))))
(count-length (cons 8 (cons 9 '())) 0)

count-medic.rkt
(layer count
  (in #:module "count.rkt"
    [(count-length)
     [on-entry
      (timeline count)
      (timeline v)
      (timeline (null? v))]]))

```

Figure 3.9. Timeline tracing

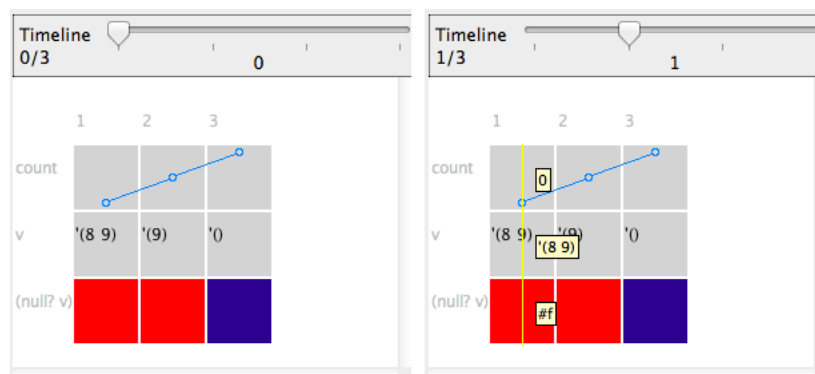


Figure 3.10. Timeline traces for timeline form

For example, with `(assert (> x 0))` and when values of `x` over time are 3, 2, 1, and 0, the assertion fails on the fourth value of `x` producing the following timeline:



Although comparisons between two elements of a trace are sometimes useful, a comparison of one trace element with its initial value is more often useful. A programmer could change (through metaprogramming) the source program to propagate the old version, but Medic makes the comparison considerably simpler through a `same?` form in a timeline trace. The `same?` form always produces true for the initial trace. Afterwards, the result is true only if the trace element's value is the same as the initial trace; the `same?` predicate compares values like Racket's `equal?`, but is extended to perform a deeper comparison by traversing opaque structures and objects. Like `(assert pred)`, only false values produced by `(same? v)` are highlighted in red in the timeline.

Clicking the *Time View* button opens a variant of the timeline view as shown in Figure 3.11. A programmer can slide through time to see which events take place at a particular time, showing not only the relative order for events of interest but also the gaps between events.

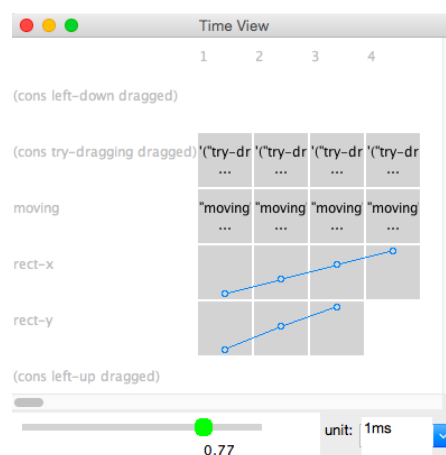


Figure 3.11. Time view

3.3 Implementation

The implementation of Medic leverages Racket’s ability to support a completely new language, plus Racket’s ability to macro-expand a program written in any language to a common core language. This combination means that Medic can offer a specialized language for writing metaprograms, and those metaprograms can introduce debugging annotations on programs written in any language that compiles to recognizable core forms, such as definitions and functions.

A debugging session starts by interpreting a Medic program, which describes debugging instructions in terms of a source program and logging instrumentation to add to the program. Interpretation extracts debugging instructions and their related debugging information, such as source location and trace layer id, into debugging tables that drive an instrumentation phase. In the instrumentation phase, logging forms are woven into the source program. The instrumented program is then compiled and run, and generated traces are piped to the front-end trace browser. The trace browser processes traces and presents them in a visual way with interactive exploration.

3.3.1 Interpretation of Medic Programs

Metaprograms written with the grammar of Figure 3.1 are interpreted in a straightforward way to construct three tables: `border-insert-table`, `at-insert-table`, and `template-table`. The first two tables describe places where debugging instrumentation is to be added, as elaborated from forms like `in`, `at`, and `each-function`, while the third table describes formatting rules for function-call logging as specified via `with-behavior`.

Each table maps a *module-name* to an *M*, *S*, or *N*, respectively, and the representations of *M*, *S*, and *N* are in Figure 3.12. A table’s content is represented by a combination of lists written as `LIST(. . . .)`, tuples written as `<. . . .>`, symbols written as identifiers prefixed by a quote mark, and syntax objects (Dybvig et al. 1993; Flatt et al. 2012) represented by the *stx* nonterminal. In particular, program fragments in a Medic program, such as logging statements to insert into the program, are represented by syntax objects.

3.3.2 Weaving Debugging Code

Similar to an AOP implementation, Medic weaves debugging instructions into the source program as described by a metaprogram. Unlike a traditional AOP implementation, the program’s source language is not fixed, but defined through the expansion of macros. Targets for weaving are

```

M ::= LIST(border-insert, ...)
border-insert ::= ⟨scope, LIST(insert, ...)⟩
insert ::= ⟨loc, LIST(stx, ...)⟩
S ::= LIST(at-insert, ...)
at-insert ::= ⟨at-scope, LIST(srcloc, ...), loc, LIST(stx, ...)⟩
N ::= LIST(tem, ...)
tem ::= ⟨name, behavior⟩
scope ::= 'module
          | 'each-function
          | 'name
at-scope ::= 'module | LIST('name, ...)
loc ::= 'entry | 'exit
srcloc ::= integer | #f
behavior ::= function behavior description

```

Figure 3.12. Debugging tables

defined syntactically, but the source code must be parsed and expanded to support the addition of code at semantically meaningful points. At the same time, identifier bindings should be resolved through a combination of the source program’s bindings and the metaprogram’s imports; for example, a use of the `log` form should refer to Medic’s `log` form, but logged variables should come from the source program.

Medic relies in part on Racket’s macro system (Flatt et al. 2012) to manage bindings and transformations. Medic expands a source program via Racket’s macro expander to produce a program in the core language, which is effectively an AST representation of the program. Source terms are correlated with expanded terms by source location, which is preserved by Racket’s macro expander. As Medic traverses a program to insert debugging forms, it tracks variable bindings, and it rewrites inserted fragments so that variables with matching names are captured by the source program’s bindings. In other words, debugging fragments are added to the program non-hygienically to enable access to variables in the source program.

When adding tracing statements, including `log`, `aggregate`, `timeline`, and `assert`, syntax properties are attached to the tracing statements to record layer and stamp information. The layer information contains the current *layer-id*, and the stamp is an integer distinguishing different tracing statements in the Medic program. For example, if two `(timeline x)` statements are inserted from two different places, they are marked with different stamp information to produce separate timeline results.

The instrumentation model starts with the syntax-object model of Flatt et al. (2012), but since weaving involves source-location and property information, Medic extends *stx* with *srcloc* and *props* pieces:

$$\begin{aligned} stx &::= \mathbf{STX}(atom, ctx, srcloc, props) \\ &\quad | \mathbf{STX}(\mathbf{LIST}(stx, \dots), ctx, srcloc, props) \\ id &::= \mathbf{STX}(sym, ctx, srcloc, props) \\ ctx &::= \text{lexical-context information} \mid \#f \\ props &::= \mathbf{LIST}(prop, \dots) \mid \#f \\ prop &::= \langle sym, val \rangle \end{aligned}$$

The *srcloc* part represents the position of a syntax object in the source program, and *props* is a sequence of key-value pairs for syntax properties.

Figure 3.13 lists some metafunctions of the model, including *convert* and *wrap-context*. The *convert* metafunction strips each debugging syntax object’s lexical context. When the *convert* metafunction encounters $\mathbf{STX}(\mathbf{LIST}(id, stx_v, \dots), ctx, srcloc, props)$ where *id* resolves to *log*, *aggregate*, *timeline*, or *assert*, layer and stamp syntax properties from the enclosing syntax object are added to the identifier. The *wrap-context* metafunction traverses a syntax object to locate identifiers whose names match entries in the *bindings* table, and when it finds a match, the identifier’s lexical context is replaced to match the binding identifier in the source. The *wrap-context* metafunction is also responsible for detecting references to function-name and replacing the function-name with the *name* of the enclosing function.

Figure 3.14 and Figure 3.15 show specific steps in Medic’s instrumentation pass. The metafunction, *weave-module*, takes the source program, which is represented as an expanded module of the form

$$\mathbf{STX}(\mathbf{LIST}(id_{module}, stx_{name}, stx_{expr}, \dots), ctx, srcloc, props),$$

as an argument. The *weave-module* metafunction locates any module-level *inserts* in *M*, and the metafunction *lookup-border-insert* extracts entry and exit additions from each *insert*. The *weave-mb* metafunction takes the exit forms to insert and adds them as part of the module body, while entry additions are added directly, since no global bindings are available for use on module entry.

The *weave-mb* metafunction transforms expressions in the module body and calls the *weave-expr* metafunction to handle expressions other than top-level definitions. For each definition of the form

$$\mathbf{STX}(\mathbf{LIST}(id_{define}, id, stx_{expr}), ctx, srcloc, props),$$

lookup-border-insert : $\text{LIST}(\text{insert}, \dots) \text{ loc } \text{LIST}(\text{stx}, \dots) \rightarrow \text{LIST}(\text{stx}, \dots)$
 Retrieve debugging code to be inserted at the border of a module or a function.

lookup-at-insert : $\text{stx } S \text{ name loc } \text{LIST}(\text{stx}, \dots) \rightarrow \text{LIST}(\text{stx}, \dots)$
 Retrieve debugging code to be inserted around an expression.

convert : $\text{stx} \rightarrow \text{stx}$
 Strip the syntax object's lexical context and add 'layer and 'stamp properties for id_{log} , $\text{id}_{\text{aggregate}}$, $\text{id}_{\text{timeline}}$ and $\text{id}_{\text{assert}}$.

wrap-context : $\text{stx bindings name} \rightarrow \text{stx}$
 Wrap identifiers within the syntax object with proper lexical context.

Figure 3.13. Retrieval and transformation of syntax objects

weave-module : $\text{stx } M \text{ } S \rightarrow \text{stx}$
 $\text{weave-module}[\text{STX}(\text{LIST}(\text{id}_{\text{module}}, \text{stx}_{\text{name}}, \text{stx}_{\text{expr}}, \dots), \text{ctx}, \text{srcloc}, \text{props}), M, S]$
 $= \text{STX}(\text{LIST}(\text{id}_{\text{module}}, \text{stx}_{\text{name}}, \text{stx}_{\text{entry}}, \dots, \text{stx}_{\text{new}}, \dots), \text{ctx}, \text{srcloc}, \text{props})$
 subject to $\text{resolve}[\text{id}_{\text{module}}] = \text{module}$,
 $\text{lookup-insert-table}[M, \text{'module}] = \text{LIST}(\text{insert}, \dots)$,
 $\text{lookup-border-insert}[\text{LIST}(\text{insert}, \dots), \text{'entry}, \epsilon] = \text{LIST}(\text{stx}_{\text{entry}}, \dots)$,
 $\text{lookup-at-insert}[\text{LIST}(\text{insert}, \dots), \text{'exit}, \epsilon] = \text{LIST}(\text{stx}_{\text{exit}}, \dots)$,
 $\text{weave-mb}[\text{LIST}(\text{stx}_{\text{expr}}, \dots), \text{LIST}(\text{stx}_{\text{exit}}, \dots), \epsilon, M, S, \epsilon] = \text{LIST}(\text{stx}_{\text{new}}, \dots)$

weave-mb : $\text{LIST}(\text{stx}, \dots) \text{ LIST}(\text{stx}, \dots) \text{ bindings } M \text{ } S \text{ LIST}(\text{stx}, \dots) \rightarrow \text{LIST}(\text{stx}, \dots)$
 $\text{weave-mb}[\epsilon, \text{LIST}(\text{stx}_{\text{exit}}, \dots), \text{bindings}, M, S, \text{LIST}(\text{stx}_{\text{result}}, \dots)] = \text{LIST}(\text{stx}_{\text{result}}, \dots, \text{stx}_{\text{exit}}, \dots)$
 subject to $\text{LIST}(\text{wrap-context}[\text{stx}_{\text{exit}}, \text{bindings}, \#f], \dots) = \text{LIST}(\text{stx}_{\text{exitc}}, \dots)$
 $\text{weave-mb}[\text{LIST}(\text{STX}(\text{LIST}(\text{id}_{\text{define}}, \text{id}, \text{stx}_{\text{expr}}), \text{ctx}, \text{srcloc}, \text{props}), \text{stx}_{\text{rest}}, \dots),$
 $\text{LIST}(\text{stx}_{\text{exit}}, \dots), \text{bindings}, M, S, \text{LIST}(\text{stx}_{\text{result}}, \dots)]$
 $= \text{weave-mb}[\text{LIST}(\text{stx}_{\text{rest}}, \dots), \text{LIST}(\text{stx}_{\text{exit}}, \dots), \text{bindings}_{\text{new}}, M, S, \text{LIST}(\text{stx}_{\text{nr}}, \dots)]$
 subject to $\text{resolve}[\text{id}_{\text{define}}] = \text{define}$, $\text{append}[\text{LIST}(\text{id}), \text{bindings}] = \text{bindings}_{\text{new}}$,
 $\text{id} = \text{STX}(\text{'name}, \text{ctx}_{\text{id}}, \text{srcloc}_{\text{id}}, \text{props}_{\text{id}})$,
 $\text{lookup-at-insert}[\text{STX}(\text{LIST}(\text{id}_{\text{define}}, \text{id}, \text{stx}_{\text{expr}}), \text{ctx}, \text{srcloc}, \text{props}),$
 $S, \#f, \text{'entry}, \epsilon] = \text{LIST}(\text{stx}_{\text{atentry}}, \dots)$,
 $\text{lookup-at-insert}[\text{STX}(\text{LIST}(\text{id}_{\text{define}}, \text{id}, \text{stx}_{\text{expr}}), \text{ctx}, \text{srcloc}, \text{props}),$
 $S, \#f, \text{'exit}, \epsilon] = \text{LIST}(\text{stx}_{\text{atexit}}, \dots)$,
 $\text{STX}(\text{LIST}(\text{id}_{\text{define}}, \text{id}, \text{weave-expr}[\text{stx}_{\text{expr}}, \text{bindings}, \text{name}, M, S]), \text{ctx}, \text{srcloc}, \text{props}) = \text{stx}_{\text{expr}}$,
 $\text{STX}(\text{'begin}, \#f, \#f, \#f) = \text{id}_{\text{begin}}$,
 $\text{STX}(\text{LIST}(\text{id}_{\text{begin}}, \text{wrap-context}[\text{stx}_{\text{atentry}}, \text{bindings}, \#f], \dots, \text{stx}_{\text{expr}},$
 $\text{wrap-context}[\text{stx}_{\text{atexit}}, \text{bindings}_{\text{new}}, \#f], \dots), \text{ctx}, \text{srcloc}, \text{props}) = \text{stx}_{\text{new}}$,
 $\text{LIST}(\text{stx}_{\text{result}}, \dots, \text{stx}_{\text{new}}) = \text{LIST}(\text{stx}_{\text{nr}}, \dots)$
 $\text{weave-mb}[\text{LIST}(\text{stx}_{\text{expr}}, \text{stx}_{\text{rest}}, \dots), \text{LIST}(\text{stx}_{\text{exit}}, \dots), \text{bindings}, M, S, \text{LIST}(\text{stx}_{\text{result}}, \dots)]$
 $= \text{weave-mb}[\text{LIST}(\text{stx}_{\text{rest}}, \dots), \text{LIST}(\text{stx}_{\text{exit}}, \dots), \text{bindings}, M, S, \text{LIST}(\text{stx}_{\text{nr}}, \dots)]$
 subject to $\text{weave-expr}[\text{stx}_{\text{expr}}, \text{bindings}, \#f, M, S] = \text{stx}_{\text{new}}$,
 $\text{LIST}(\text{stx}_{\text{result}}, \dots, \text{stx}_{\text{new}}) = \text{LIST}(\text{stx}_{\text{nr}}, \dots)$

Figure 3.14. Process of weaving module-level code

weave-expr : $stx\ bindings\ name\ M\ S \rightarrow stx$

weave-expr[[stx_{fun} , $bindings$, $name$, M , S]]
 $= stx_{new}$
 subject to $stx_{fun} = \text{STX}(\text{LIST}(id_{lambda}, \text{STX}(\text{LIST}(id_a, \dots), ctx_a, srcloc_a, props_a), stx_{expr}, \dots), ctx, srcloc, props),$
 $\text{resolve}[[id_{lambda}]] = \text{lambda}, \text{append}[[\text{LIST}(id_a, \dots), bindings]] = bindings_{new},$
 $\text{lookup-insert-table}[[M, 'name]] = \text{LIST}(insert_f, \dots),$
 $\text{lookup-insert-table}[[M, 'each-function]] = \text{LIST}(insert_{ef}, \dots),$
 $\text{append}[[\text{LIST}(insert_f, \dots), \text{LIST}(insert_{ef}, \dots)]] = \text{LIST}(insert, \dots),$
 $\text{lookup-border-insert}[[\text{LIST}(insert, \dots), 'entry, \epsilon]] = \text{LIST}(stx_{fentry}, \dots),$
 $\text{lookup-border-insert}[[\text{LIST}(insert, \dots), 'exit, \epsilon]] = \text{LIST}(stx_{fexit}, \dots),$
 $(\text{weave-expr}[[stx_{expr}, bindings_{new}, name, M, S] \dots]) = (stx_{exprc} \dots),$
 $\text{STX}('begin0, \#f, \#f, \#f) = id_{begin0}, \text{STX}('begin, \#f, \#f, \#f) = id_{begin},$
 $\text{STX}(\text{LIST}(id_{lambda},$
 $\quad \text{STX}(\text{LIST}(id_a, \dots), ctx_a, srcloc_a, props_a),$
 $\quad \text{wrap-context}[[stx_{fentry}, bindings_{new}, name]], \dots,$
 $\quad \text{STX}(\text{LIST}(id_{begin0}, \text{STX}(\text{LIST}(id_{begin}, stx_{exprc}, \dots), \#f, \#f, \#f),$
 $\quad \quad \text{wrap-context}[[stx_{fexit}, bindings_{new}, name]], \dots), \#f, \#f, \#f)),$
 $\quad ctx, srcloc, props) = stx_{fb}$
 $\text{lookup-at-insert}[[stx_{fun}, S, name, 'entry, \epsilon]] = \text{LIST}(stx_{atentry}, \dots),$
 $\text{lookup-at-insert}[[stx_{fun}, S, name, 'exit, \epsilon]] = \text{LIST}(stx_{atexit}, \dots),$
 $\text{STX}(\text{LIST}(id_{begin},$
 $\quad \text{wrap-context}[[stx_{atentry}, bindings, name]], \dots, stx_{fb},$
 $\quad \text{wrap-context}[[stx_{atexit}, bindings, name]], \dots), ctx, srcloc, props) = stx_{new}$

Figure 3.15. Process of weaving expression-level code

a new binding of id is added to $bindings$, and any possible insertion of debugging code around the definition is found through `lookup-at-insert`, which calls `convert` before returning results. The definition is augmented with any existing $stx_{atentry}S$ and $stx_{atexit}S$ (with `wrap-context` called first) to make a sequence of syntax objects. The `weave-expr` metafunction is applied to the body of the definition, which uses $name$ obtained from id for later run-time function scope lookup.

Since *border-insert* involves function-level insertions of debugging code, we primarily explain the stx_{fun} case in the `weave-expr` metafunction, while other stx cases just need to consider expression-level insertions by checking the S table. The list of $bindings$ is extended with the function's formal parameters. A sequence of *inserts* is obtained by finding mappings with the current function-name scope, *'name*, and *'each-function* scope. If there are no $stx_{fentry}S$ or $stx_{fexit}S$, `wrap-context` will not be called, and no debugging expressions will be inserted. Otherwise, the transformed function body, stx_{fb} , has the shape

```
(lambda (arg ...)
  body-entry-expr ...
  (begin0 (begin body-expr ...)
    body-exit-expr ...))
```

Expressions inside a `begin0` form are evaluated in order, but the result of the `begin0` form is the result of the first expression, so `begin0` keeps the original function’s return value intact. Finally, stx_{fun} is transformed into stx_{new} , which is stx_{fb} wrapped with expression-level insertions, if any.

3.3.3 Generation and Presentation of Traces

Macro transformations are performed on tracing statements to generate appropriate raw traces at run time for the trace browser’s display.

For the `log` forms, the macro transformer dispatches on the shape of the `log` form. For `(log id)`, it produces a result of “*id-name* = *id-value*” where *id-name* and *id-value* are replaced by the identifier’s name and value. For the `(log (fun arg ...))` case, Medic looks up any declared `with-behavior` for the current function, *fun*, in the template table, *N*. If there is a behavior template defined for *fun*, `log` produces a result for the behavior template by substituting the result of evaluating `@expr` for any occurrence of `@expr` in the behavior template; otherwise, the value of `(fun arg ...)` is augmented with extra context information. For the `(log form v ...)` case, where *form* is a string containing $\sim a$, `log` substitutes the value from the *vs* corresponding to the position of the $\sim a$. When the `log` is prefixed with an `@` notation, `@function-name` or `@expr` is substituted with its run-time value. Other cases of `log` return the literal values. A table records the result of the `log` statement, its corresponding *layer-id* (obtained from the syntax object’s *layer* property), and if the trace is a behavior defined by `with-behavior`.

For the `node` and `edge` forms, Medic just records the arguments and stores the node-associated and edge-associated data in two separate tables. The `remove-node` and `remove-edge` forms delete the corresponding node and edge in these two tables. The `aggregate`, `timeline`, and `assert` forms are implemented to emit an event containing expression labels, values, execution time, and stamp information where the stamp is obtained from each syntax object’s *stamp* property. Data with the same stamp are accumulated in the same entry of a table. For `(same? v)`, the *v* label, value, and execution time are recorded.

The trace browser displays different kinds of traces in its associated pane. Log traces, aggregate traces, and timeline traces (excluding timeline traces produced by `same?` forms) are displayed linearly in recorded order. For graph traces, each node entity *n* from `(node n)` should be comparable and show object identity. A node in the graph pane is created for *n*, and the same node entity *n* cannot produce multiple nodes despite repeated evaluation of `(node n)`. The `(edge a b)`

statement adds an edge connecting a to b . If there is no node corresponding to a or b , a new node will be created. As for $(\text{same? } v)$, equality of all the primitive data members of v to the initial state is recursively checked. If v is a class, all of its public, private, and inherited fields are checked to see if the state of v changes over time.

CHAPTER 4

DEBUGGING WITH DOMAIN-SPECIFIC EVENTS

In Racket, a DSL is created by making extensions to existing languages, and macros provide high-level support for defining new language extensions. After examining the advantages and disadvantages of different debugging techniques presented in Chapter 2 and after hands-on experience with trace-oriented debugging in Chapter 3, the research adopted an event-based debugging model. There are two reasons. First, an event-based view allows each language to capture run-time states and evaluation rules and enables reuse and extension of a debugger. Second, an event-based view meshes well with a macro-expansion view of language implementation where macros can be paired with events to capture domain concepts. This chapter presents an approach to debugging DSLs that maps DSLs to domain-specific events. A variant of the chapter was originally published as *Debugging with Domain-Specific Events via Macros* (Li and Flatt 2017) in the Software Language Engineering conference, <https://doi.org/10.1145/3136014.3136019>.

4.1 Motivation

To see the need for domain-specific events to implement a DSL debugger, consider the case of POP-PL (Florence et al. 2015). POP-PL is a “patient-oriented prescription programming language” that is meant to enable a doctor to describe and automate a course of treatment. The language is message-based, where a message might correspond to adjusting a medical device or calling a nurse to take a specific action. POP-PL is implemented by macro expansion to conventional functional and imperative programming constructs. If we try to rely on the underlying GPL’s stepping-based debugger and map POP-PL source terms to those debugging events, the debugger would not match a health-care professional’s view of the computation. In fact, the execution flow of POP-PL programs cannot be easily controlled by setting breakpoints and stepping to suspend and resume execution. Instead, the execution flow of a POP-PL program is meant to be indefinite and depend on the new incoming messages in the network and available prescription handlers.

A debugger for POP-PL should instead present debugging in terms of the events that capture domain concepts. For example, the following is a POP-PL program named “heparin.pop” that contains three handles, `initially`, `infusion`, and `aPTTChecking`, to react with messages:

```
#lang pop-pl

used by JessieBrownVA

initially
  giveBolus 80 units/kg of: HEParin by: iv
  start 18 units/kg/hour of: HEParin by: iv

infusion:
  whenever new aPTTResult
    aPTT < 45      | giveBolus 80 units/kg of: HEParin by: iv
                   | increase HEParin by: 3 units/kg/hour
    aPTT > 123     | hold HEParin
                   | after 1 hour
                   |   restart HEParin
                   |   decrease HEParin by: 4 units/kg/hour

aPTTChecking:
  every 6 hours checkaPTT whenever aPTTResult outside of 59 to 101, x2
  every 24 hours checkaPTT whenever aPTTResult in range 59 to 101, x2
```

For the above POP-PL program, we have a debugging interface as shown in Figure 4.1. In the interface, the `checkaptt` message entry was clicked in the top-right Messages window, which triggered a display of the handler information in the bottom-right window. Clicking a message in the bottom-right window brought navigation to the source context on the left, which was highlighted in yellow. The POP-PL debugger updates the Messages window when a new message arrives in the system and when messages are sent from handlers in response to arriving messages. The debugger supports graphical navigation from a message to a handler and from a message to the source code as well. We should use domain-specific events such as a `receive-msg-e` event to capture the concept of an arriving message and a `send-msg-e` event to capture the concept of an outgoing message from handlers.

Figure 4.2 offers a diagram to show the difference between the traditional approach of building on a GPL debugger and my approach to DSL debugger construction. The traditional approach tries to reuse the GPL debugger support, resulting in traditional, stepping-based debugging techniques for DSLs. In comparison, I aim for domain-specific debuggers where the debugger only relies on the DSL events without the necessity of a GPL debugger and where a DSL can reuse low-level

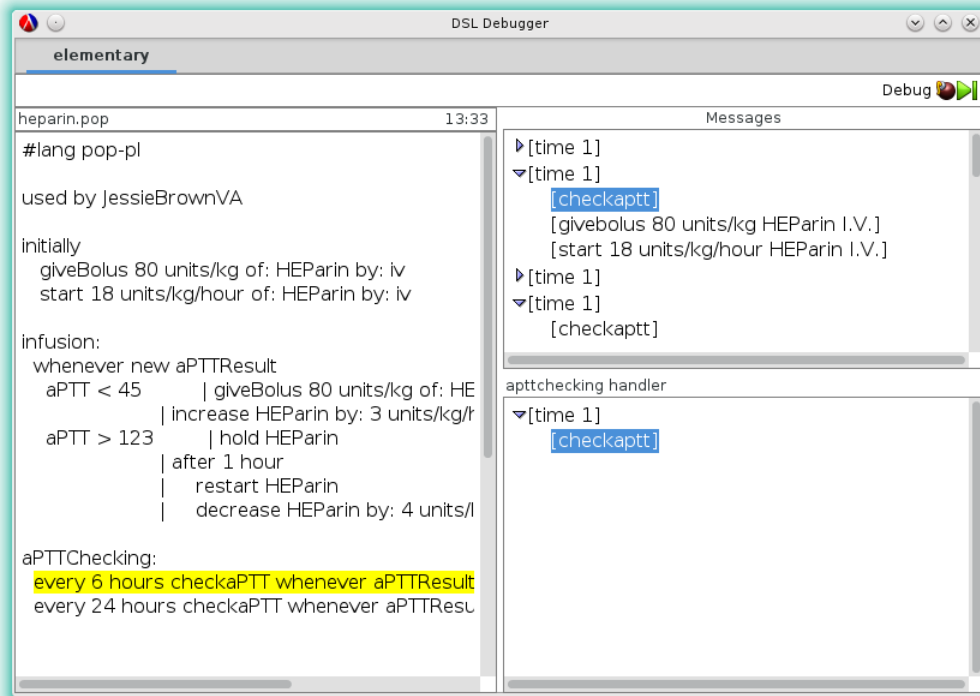


Figure 4.1. POP-PL debugger

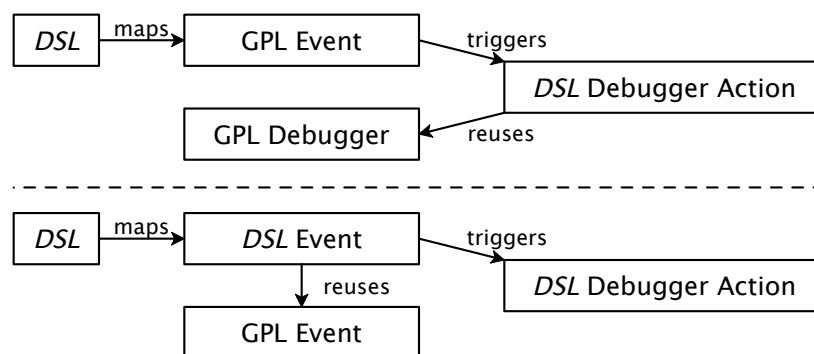


Figure 4.2. A comparison of the traditional approach (top) and my approach (bottom) to DSL debugger construction

events to define domain-specific events.

4.2 Implementing DSLs with Macros

The approach of mapping general-purpose events to domain-specific events fits together naturally with an evaluation approach that maps DSL programs into GPL programs via macros. In Racket specifically, building a DSL typically involves a *reader-level extension* to parse a DSL program into parenthesized forms (S-expressions), and then an *expander-level extension* that relies on macros to expand the parenthesized forms.

A module in Racket is both a unit of compilation and the mechanism for organizing macros and language layers. In the simplest case, a DSL program resides in its own module, while the implementation of the DSL itself resides in another module that is referenced by the DSL program. The DSL implementation is written in some language as defined by module imports, and the DSL’s macros expand into the imported constructs—where the imports can be macros that expand to another language, and so on.

Figure 4.3 illustrates the overall process of expanding a single DSL module via macros and through multiple language layers. (The figure does not show the implementations of the layers, but only the way that the original module expands into each layer.) Box 1 shows a program written in a toy DSL called `point`. The `point` language contains initialization statements for the `x` and `y` properties and operations such as `move x by` and `move y by` to manipulate the two properties. The `#lang point` declaration in the box selects the reader that parses the program into a `define-point` S-expression, which is sketched in box 2. In addition to that S-expression, box 2 must import a `module-begin` macro (referred to as “macro 1” among the figure’s arrows) and a `define-point` macro (referred to as “macro 2”). The `module-begin` macro’s job is to add a definition of `point` to the beginning of the module, and the `define-point` macro’s job is to make an instance of `point` and call some of its methods.

To a first approximation, each of those macros can be implemented as a simple pattern-based macro, which uses the form

```
(define-syntax-rule pattern template)
```

to indicate that each instance of *pattern* should be replaced by an instance of *template*. Pattern variables bound in *pattern* are replaced as *template* is instantiated. So, `module-begin` and

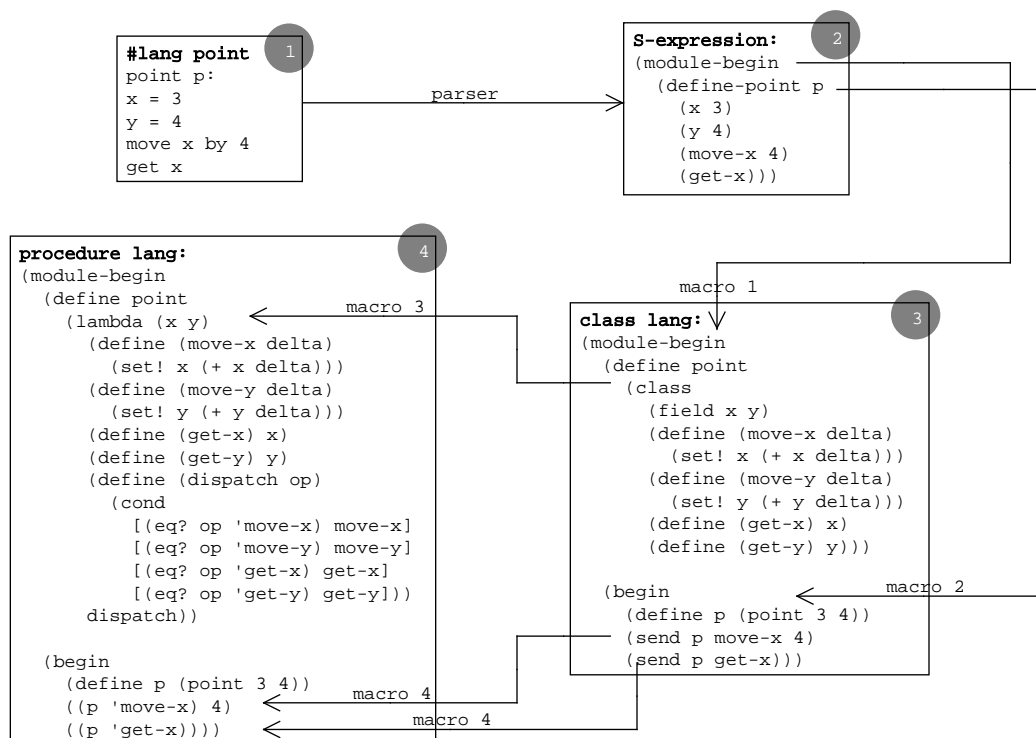


Figure 4.3. Macro expansion

`define-point` also can be defined as

```
(define-syntax-rule (module-begin decl ...)
  (base-module-begin
    ; add a point class declaration
    (define point
      (class
        details omitted, but see box 3 in the figure))
    decl ...))

(define-syntax-rule (define-point name (x x-expr) (y y-expr)
  (op arg ...) ...)
  (begin
    (define name (point x-expr y-expr))
    (send name op arg ...) ...))
```

The pattern `(module-begin decl ...)` matches any term that starts with `module-begin` followed by any number of terms bound to the pattern variable `decl`. The `...` after `decl` causes `decl` to stand for zero or more matches. The expansion of the macro is a `base-module-begin` form that defines the name `point` and continues with all the supplied `decls`. Similarly, the pattern for `define-point` matches that name followed by at least two terms, where `x-expr` stands for the second part of the first term, `y-expr` stands for the second part of the second term, `op` stands for the called method in each subsequent term, and `arg` stands for the arguments of each of the called methods (i.e., `arg` is a list of lists). Note that the number of `...`s after a pattern variable in a template matches the number of `...`s after the same pattern variable in the pattern. The `base-module-begin`, `define`, `class` and `send` forms used in the macro expansion are all imported into the module that defines the macros.

Although simple pattern-based macros work for many cases, these macros are not quite right for `module-begin` and `define-point`. The `define-point` macro wants `x` and `y` to be literally the identifiers `x` and `y`, instead of pattern variables that match any term. In addition, the macros `module-begin` and `define-point` independently introduce a definition and references to `point`, so macro *hygiene* keeps them separate (Kohlbecker et al. 1986) instead of shared as intended.

To solve these problems, `module-begin` and `define-point` are rewritten as general compile-time functions with `syntax-case` forms that help with pattern matching a template instantiation:

```
(define-syntax id (lambda (source-expr)
  (syntax-case source-expr (literal-id ...)
    [pattern optional-guard-expr template-expr] ...)))
```

The `module-begin` definition above can be rewritten as

```
(define-syntax module-begin
  (lambda (stx)
    (syntax-case stx ()
      [(-)
       #'(base-module-begin)] ; no decls ⇒ no class
      [(_ decl ...)
       #'(base-module-begin
          (define point details omitted)
          decl ...))]))
```

where `define-syntax` binds `module-begin` to a compile-time function that receives a representation `stx` of the macro use. The `syntax-case` form dispatches on that `stx` to match one of the subsequent patterns; I add a new pattern here, as a kind of optimization, to drop the definition of `point` if there are no `decls` to use it. After matching a pattern in `syntax-case`, the corresponding clause can perform arbitrary compile-time work, but `#'` produces a compile-time value from a template instantiation, just like a simple pattern-matching macro.

From now on, I will abbreviate a definition

```
(define-syntax id (lambda (arg-id) body-expr))
⇒
(define-syntax (id arg-id) body-expr)
```

To fix `module-begin`, the `point` identifier in the template needs to be replaced with `point` as if it appeared in the macro-use site. The expression `(syntax-local-introduce #'point)` generates such an identifier,¹ and I can inject it into the pattern world using `with-syntax`, which binds a pattern to the result of a compile-time expression:

```
(define-syntax (module-begin stx)
  (syntax-case stx ()
    [(-)
     #'(base-module-begin)]
    [(_ decl ...)
     (with-syntax ([point-id (syntax-local-introduce #'point)])
       #'(base-module-begin
          (define point-id details omitted)
          decl ...))]))
```

The repair to `define-point` is similar, but also uses the parentheses that appear after the first argument to `syntax-case`, which hold identifiers to be treated as literals instead of pattern variables:

¹Using `syntax-local-introduce` is rarely the best strategy, but it suffices for the example here.

```
(define-syntax (define-point stx)
  (syntax-case stx (x y)
    [(_ name (x x-expr) (y y-expr) (op arg ...) ...)
     (with-syntax ([point-id (syntax-local-introduce #'point)])
       #'(begin
            (define name (point-id x-expr y-expr))
            (send name op arg ...) ...)))]))
```

The expansion of the `module-begin` and `define-point` macros on the code in box 2 of Figure 4.3 produces the code in box 3 of the figure. Box 3 shows another `module-begin` in place of `base-module-begin` on the assumption that the form imported as `base-module-begin` by the macro-implementing module is exported from its defining module as `module-begin`.

For the language of box 3, assume that `module-begin` adds nothing to its content, and consider further the `class` and `send` macros that must be imported there. The `class` macro implements a class abstraction in terms of procedures, and the `send` macro accordingly transforms method calls into nested procedure calls.

```
(define-syntax (class stx)
  (syntax-case stx (field define)
    [(class (field f ...)
            (define (method-name arg ...) expr ...) ...)
     (andmap identifier?
              (syntax->list #'(method-name ... arg ... ...)))
     #'(lambda (f ...)
         (define (method-name arg ...) expr ...) ...
         (define (dispatch op)
           (cond
            [(eq? op 'method-name) method-name] ...))
         dispatch)))]))
(define-syntax (send stx)
  (syntax-case stx ()
    [(send obj method-name arg ...)
     (identifier? #'method-name)
     #'((obj 'method-name) arg ...)]))
```

That is, a `class` form turns into a function that accepts field values for an instance and returns a dispatch function for the instance, and a `send` form turns into a call of that dispatch function passing the method name as a symbol. These macro implementations contain additional compile-time code as guards to check that each method name and method argument is an identifier (as opposed to, say, a number) before generating the expansion, which illustrates another use of compile-time computation.

Using the `class` and `send` forms, the original DSL program is further expanded into the forms in box 4. If the procedure language is created through more macro transformations into some other existing language, then the expansion of the DSL program requires additional steps. The general case of a module expansion is depicted on the left-hand side of Figure 4.4, where a DSL is eventually compiled to a core language, and each dotted line represents the macro transformations in a compilation step.

The right-hand side of Figure 4.4 is the contribution of this chapter. I define a set of events that are produced by the evaluation of a program in the core language, and I show how to enrich and complement the macro-expansion steps on the left-hand side with event-mapping rules for the right-hand side.

4.3 Core Events

The core language in the bottom left of Figure 4.4 can be any simple programming language. For Racket, the core language is a variant of the λ -calculus with primitives and mutable variables, so a CESK machine (Felleisen and Friedman 1987) can model the core language. A CESK machine explicitly manages a lexical environment, continuation, and store, so it serves as a general model of a GPL that exposes features relevant for debugging—but it has no notion of *events*.

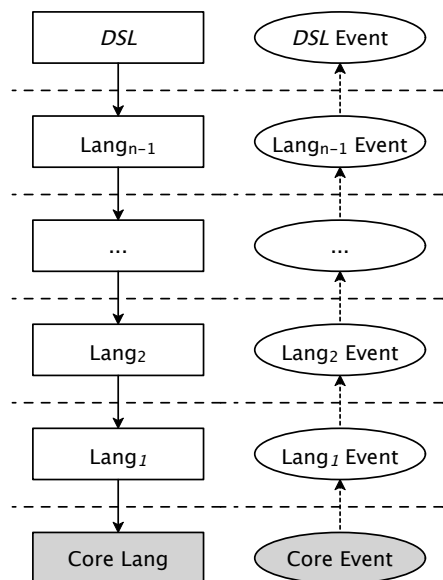


Figure 4.4. An overview of layered DSL implementation and domain-specific event support

Figure 4.5 defines an extension of the CESK machine that includes a slot for an event trace, and Figure 4.6 shows the associated grammars. Each step in the *CESKT* machine adds one or more events to the trace component of the machine. The resulting trace models the sequence of events that a debugger can receive to report on the progress of the computation. The trace component is similar to the time component in the time-stamped CESK machine for static analyses (Van Horn and Might 2011) but with an emphasis on capturing evaluation details to report debugging information.

An *event* is modeled as some interesting point in the dynamic execution of a program. There are five kinds of core events: `construct-e`, `function-e`, `variable-update-e`, `cont-add-e`, and `cont-rmv-e`. Evaluation of a program generates event instances of these event classes, and each kind of event carries event-specific debugging information. The `updt-t` metafunction packages event-specific information into new events:

$$\begin{aligned}
 \text{updt-t}[T, N, \langle \text{construct-e}, M, \mathcal{E} \rangle] &= \langle \text{evnt}, T \rangle \\
 \text{subject to } \{ (\text{expression } M) (\text{bindings } \mathcal{E}) \} &= A, \\
 \langle \text{construct-e}, N, \text{core}, A \rangle &= \text{evnt} \\
 \text{updt-t}[T, N, \langle \text{store-e}, X, \langle \sigma, \langle V, \mathcal{E} \rangle \rangle \rangle] &= \langle \text{evnt}, T \rangle \\
 \text{subject to } \{ (\text{name } X) (\text{slot } \sigma) (\text{value } \langle V, \mathcal{E} \rangle) \} &= A, \\
 \langle \text{store-e}, N, \text{core}, A \rangle &= \text{evnt} \\
 \text{updt-t}[T, N, \langle \text{cont-add-e}, \text{cont} \rangle] &= \langle \text{evnt}, T \rangle \\
 \text{subject to } \{ (\text{continuation } \text{cont}) \} &= A, \\
 \langle \text{cont-add-e}, N, \text{core}, A \rangle &= \text{evnt} \\
 \text{updt-t}[T, N, \langle \text{cont-rmv-e} \rangle] &= \langle \text{evnt}, T \rangle \\
 \text{subject to } \langle \text{cont-rmv-e}, N, \text{core}, \emptyset \rangle &= \text{evnt} \\
 \text{updt-t}[T, N, et, et_{rest}, \dots] &= T_{new} \\
 \text{subject to } \text{updt-t}[T, N, et] &= T_l, \\
 \text{updt-t}[T_l, N, et_{rest}, \dots] &= T_{new}
 \end{aligned}$$

Each event instance is a data structure containing its event class name, source information, an event tag indicating the event's origin in the core language, and an event attribute mapping (*A*).

Every reduction rule in Figure 4.5 starts with a `construct-e` event to reflect the occasion of a single-step reduction of an expression. The `function-e` event and the `variable-update-e` event capture the store changes in the `[apply]` and `[assign]` rules. To monitor the continuation changes of the CESK machine, two continuation-related events are included: `cont-add-e` and `cont-rmv-e`.

The trace does not contain all of the information that will be needed to construct a *domain-specific* view of the computation, however. Although domain-specific details can be encoded in

$$\begin{array}{ll}
\langle\langle (MN), \mathcal{E} \rangle, \Sigma, K, T \rangle \longrightarrow & \text{[to-apply]} \\
\langle\langle M, \mathcal{E} \rangle, \Sigma, \langle \text{ar}, \langle N, \mathcal{E} \rangle, K \rangle, T_{new} \rangle & \\
\text{subject to } \text{updt-t}[T, (MN), \langle \text{construct-e}, M, \mathcal{E} \rangle, \langle \text{cont-add-e}, \langle \text{ar}, \langle N, \mathcal{E} \rangle \rangle \rangle] = T_{new} & \\
\langle\langle (o MN \dots), \mathcal{E} \rangle, \Sigma, K, T \rangle \longrightarrow & \text{[to-prim]} \\
\langle\langle M, \mathcal{E} \rangle, \Sigma, \langle \text{op}, \langle o \rangle, \langle\langle N, \mathcal{E} \rangle, \dots \rangle, K \rangle, T_{new} \rangle & \\
\text{subject to } \text{updt-t}[T, (o MN \dots), & \\
\langle \text{construct-e}, M, \mathcal{E} \rangle, \langle \text{cont-add-e}, \langle \text{op}, \langle o \rangle, \langle\langle N, \mathcal{E} \rangle, \dots \rangle \rangle \rangle] = T_{new} & \\
\langle\langle V, \mathcal{E} \rangle, \Sigma, \langle \text{fn}, \langle (\lambda X M), \mathcal{E}_I \rangle, K \rangle, T \rangle \longrightarrow & \text{[apply]} \\
\langle\langle M, \mathcal{E}_I[X \leftarrow \sigma] \rangle, \Sigma[\sigma \leftarrow \langle V, \mathcal{E} \rangle], K, T_{new} \rangle & \\
\text{subject to } V \notin X, \sigma \notin \text{dom}(\Sigma), \text{updt-t}[T, V, \langle \text{construct-e}, M, \mathcal{E}_I \rangle, & \\
\langle \text{function-e}, X, \langle \sigma, \langle V, \mathcal{E} \rangle \rangle \rangle, \langle \text{cont-rmv-e} \rangle] = T_{new} & \\
\langle\langle V, \mathcal{E} \rangle, \Sigma, \langle \text{ar}, \langle N, \mathcal{E}_N \rangle, K \rangle, T \rangle \longrightarrow & \text{[apply-arg]} \\
\langle\langle N, \mathcal{E}_N \rangle, \Sigma, \langle \text{fn}, \langle V, \mathcal{E} \rangle, K \rangle, T_{new} \rangle & \\
\text{subject to } V \notin X, \text{updt-t}[T, V, \langle \text{construct-e}, N, \mathcal{E}_N \rangle, & \\
\langle \text{cont-rmv-e} \rangle, \langle \text{cont-add-e}, \langle \text{fn}, \langle V, \mathcal{E} \rangle \rangle \rangle] = T_{new} & \\
\langle\langle b_m, \mathcal{E} \rangle, \Sigma, \langle \text{op}, \langle\langle b_{rest}, \mathcal{E}_{rest} \rangle, \dots, \langle b_I, \mathcal{E}_I \rangle, o \rangle, \langle \rangle, K \rangle, T \rangle \longrightarrow & \text{[prim]} \\
\langle\langle b, \emptyset \rangle, \Sigma, K, T_{new} \rangle & \\
\text{subject to } \delta(o, b_I, b_{rest}, \dots, b_m) = b, \text{updt-t}[T, b_m, \langle \text{construct-e}, b, \emptyset \rangle, \langle \text{cont-rmv-e} \rangle] = T_{new} & \\
\langle\langle V, \mathcal{E} \rangle, \Sigma, \langle \text{op}, \langle v_R, \dots, o \rangle, \langle\langle N, \mathcal{E}_N \rangle, v_L, \dots \rangle, K \rangle, T \rangle \longrightarrow & \text{[prim-arg]} \\
\langle\langle N, \mathcal{E}_N \rangle, \Sigma, \langle \text{op}, \langle\langle V, \mathcal{E} \rangle, v_R, \dots, o \rangle, \langle v_L, \dots \rangle, K \rangle, T_{new} \rangle & \\
\text{subject to } V \notin X, \text{updt-t}[T, V, \langle \text{construct-e}, N, \mathcal{E}_N \rangle, \langle \text{cont-rmv-e} \rangle, & \\
\langle \text{cont-add-e}, \langle \text{op}, \langle\langle V, \mathcal{E} \rangle, v_R, \dots, o \rangle, \langle v_L, \dots \rangle \rangle \rangle] = T_{new} & \\
\langle\langle X, \mathcal{E} \rangle, \Sigma, K, T \rangle \longrightarrow & \text{[var]} \\
\langle\langle V, \mathcal{E}_V \rangle, \Sigma, K, T_{new} \rangle & \\
\text{subject to } \Sigma(\mathcal{E}(X)) = \langle V, \mathcal{E}_V \rangle, \text{updt-t}[T, X, \langle \text{construct-e}, V, \mathcal{E}_V \rangle] = T_{new} & \\
\langle\langle (\text{set } X M), \mathcal{E} \rangle, \Sigma, K, T \rangle \longrightarrow & \text{[to-assign]} \\
\langle\langle M, \mathcal{E} \rangle, \Sigma, \langle \text{set}, \langle X, \mathcal{E} \rangle, K \rangle, T_{new} \rangle & \\
\text{subject to } \text{updt-t}[T, (\text{set } X M), \langle \text{construct-e}, M, \mathcal{E} \rangle, \langle \text{cont-add-e}, \langle \text{set}, \langle X, \mathcal{E} \rangle \rangle \rangle] = T_{new} & \\
\langle\langle V, \mathcal{E} \rangle, \Sigma, \langle \text{set}, \langle X, \mathcal{E}_X \rangle, K \rangle, T \rangle \longrightarrow & \text{[assign]} \\
\langle\langle \Sigma(\sigma), \Sigma[\sigma \leftarrow \langle V, \mathcal{E} \rangle], K, T_{new} \rangle & \\
\text{subject to } V \notin X, \mathcal{E}_X(X) = \sigma, \Sigma(\sigma) = \langle V_I, \mathcal{E}_I \rangle, \text{updt-t}[T, V, \langle \text{construct-e}, V_I, \mathcal{E}_I \rangle, & \\
\langle \text{variable-update-e}, X, \langle \sigma, \langle V, \mathcal{E} \rangle \rangle \rangle, & \\
\langle \text{cont-rmv-e} \rangle] = T_{new} &
\end{array}$$

Figure 4.5. The CESKT machine

$$\begin{aligned}
M, N, L &::= V \\
&\quad | (MM) \\
&\quad | (o\ M\ M\ \dots) \\
&\quad | (\text{set } X\ M) \\
o &::= + \mid - \mid * \mid \text{add1} \mid \text{sub1} \mid \dots \\
V &::= X \mid (\lambda X\ M) \mid b \mid \text{bool} \\
\text{bool} &::= \text{true} \mid \text{false} \\
b &::= \text{number} \\
v &::= \langle V, \mathcal{E} \rangle \\
\mathcal{E} &::= \emptyset \mid \{ (X\ \sigma) \dots \} \\
B &::= \emptyset \mid \{ \langle \sigma, v \rangle \dots \} \\
\Sigma &::= \emptyset \mid \{ \langle \sigma, v \rangle \dots \} \\
A &::= \emptyset \mid \{ (\text{name attr-val}) \dots \} \\
T &::= \text{mt} \mid \langle \text{evnt}, T \rangle \\
\sigma &::= \text{a location} \\
\text{store-e} &::= \text{function-e} \\
&\quad \mid \text{variable-update-e} \\
X &::= \text{variable-not-otherwise-mentioned}
\end{aligned}$$

Figure 4.6. Grammars for the extended CESK machine

aspects of the CESKT machine (analogous to encoding numbers as Church numerals), a more direct and useful approach is to add an extra instruction to the machine to support the logging of arbitrary events:

$$\begin{aligned}
\langle \langle (\text{core-emit } \text{ename } L\ A), \mathcal{E} \rangle, \Sigma, K, T \rangle &\longrightarrow \langle \langle \text{true}, \emptyset \rangle, \Sigma, K, \langle \text{evnt}, T \rangle \rangle \\
&\text{subject to } \langle \text{ename}, L, \text{core}, A \rangle = \text{evnt}
\end{aligned}$$

In fact, this rule makes the logging parts of all other CESKT rules redundant in the sense that a source program can be instrumented with `core-emit` forms to generate exactly the events that the other rules would record in the event trace. The instrumentation metafunction, `inst`, instruments the input program, M , to generate core events. The `inst` metafunction first adds a X_e binding to record the run-time environment:

$$\text{inst} \llbracket M \rrbracket = (\text{let } ([X_e\ \emptyset])\ \text{inst-emit} \llbracket M \rrbracket)$$

The `inst-emit` metafunction instruments the program with event-generating `core-emit` terms:

$$\begin{aligned}
&[\text{inst1}]: \\
&\text{inst-emit} \llbracket X \rrbracket = (\text{begin} \\
&\quad (\text{core-emit construct-e } X\ \{ (\text{expression } V)\ (\text{bindings } \mathcal{E}) \}) \\
&\quad X) \\
&\text{subject to } \text{eval} \llbracket X \rrbracket = \langle V, \mathcal{E} \rangle
\end{aligned}$$

[inst2]:
 inst-emit $\llbracket (\lambda X M) \rrbracket =$
 $(\lambda X$
 $(\text{let } ([X_e X_e [X \leftarrow 'X]])$
 $(\text{core-emit construct-e } L$
 $\{ (\text{expression } M)$
 $(\text{bindings } \mathcal{E}) \})$
 $(\text{core-emit function-e } L$
 $\{ (\text{name } X)$
 $(\text{slot } 'X)$
 $(\text{value eval}\llbracket X \rrbracket) \})$
 $(\text{core-emit cont-rmv-e } L \emptyset)$
 $\text{inst-emit}\llbracket M \rrbracket)$)
 subject to $(\lambda X M) = L,$
 $\text{eval}\llbracket X_e \rrbracket = \mathcal{E}$

[inst3]:
 inst-emit $\llbracket V \rrbracket =$
 V

[inst4]:
 inst-emit $\llbracket (M N) \rrbracket =$
 $(\text{let } ([X_m (\text{begin}$
 $(\text{core-emit construct-e } (M N)$
 $\{ (\text{expression } M)$
 $(\text{bindings } \mathcal{E}) \})$
 $(\text{core-emit cont-add-e } (M N)$
 $\{ (\text{continuation } \langle \text{ar}, \langle N, \mathcal{E} \rangle \rangle) \})$
 $\text{inst-emit}\llbracket M \rrbracket)])$
 $(X_m (\text{begin}$
 $(\text{core-emit construct-e } M$
 $\{ (\text{expression } N)$
 $(\text{bindings } \mathcal{E}) \})$
 $(\text{core-emit cont-rmv-e } M \emptyset)$
 $(\text{core-emit cont-add-e } M$
 $\{ (\text{continuation } \langle \text{fn}, \text{eval}\llbracket X_m \rrbracket \rangle) \})$
 $\text{inst-emit}\llbracket N \rrbracket)])$)
 subject to $\text{eval}\llbracket X_e \rrbracket = \mathcal{E},$
 $M = (\lambda X L)$

[inst5]:
 inst-emit $\llbracket (o M N \dots) \rrbracket =$
 $(\text{let } ([X_m (\text{begin}$
 $(\text{core-emit construct-e } L$
 $\{ (\text{expression } M)$
 $(\text{bindings } \mathcal{E}) \})$
 $(\text{core-emit cont-add-e } L$
 $\{ (\text{continuation } \langle \text{op}, \langle o \rangle, \langle \langle N, \mathcal{E} \rangle, \dots \rangle \rangle) \})$
 $\text{inst-emit}\llbracket M \rrbracket)])$
 $\text{inst-op}\llbracket \text{eval}\llbracket X_m \rrbracket, \langle o \rangle, \langle \langle N, \mathcal{E} \rangle, \dots \rangle \rrbracket)$
 subject to $(o M N \dots) = L,$
 $\text{eval}\llbracket X_e \rrbracket = \mathcal{E}$

[inst6]:

$$\text{inst-emit}[(\text{set } X \ M)] = (\text{let } ([v_{old} \ \text{eval}[[X]]]) \\
(\text{let } ([X_{new} \ (\text{begin} \\
(\text{core-emit construct-e } L \\
\{(\text{expression } M) \ (\text{bindings } \mathcal{E})\}) \\
(\text{core-emit cont-add-e } L \\
\{(\text{continuation } \langle \text{set}, \langle X, \mathcal{E} \rangle \rangle)\}) \\
\text{inst-emit}[[M]])]) \\
(\text{core-emit construct-e } V_{new} \\
\{(\text{expression } V_{old}) \ (\text{bindings } \mathcal{E}_{old})\}) \\
(\text{core-emit variable-update-e } V_{new} \\
\{(name \ X) \ (\text{slot } 'X) \ (\text{value } \langle V_{new}, \mathcal{E}_{new} \rangle)\}) \\
(\text{core-emit cont-rmv-e } V_{new} \ \emptyset) \\
X_{old}))$$
subject to $(\text{set } X \ M) = L$,
 $\text{eval}[[X_e]] = \mathcal{E}$,
 $\text{eval}[[X_{new}]] = \langle V_{new}, \mathcal{E}_{new} \rangle$,
 $v_{old} = \langle V_{old}, \mathcal{E}_{old} \rangle$

The definition for the inst-op metafunction is:

$$\text{inst-op}[\langle b_m, \mathcal{E} \rangle, \langle \langle b_r, \mathcal{E}_r \rangle, \dots, \langle b_l, \mathcal{E}_l \rangle, o \rangle, \langle \rangle] = \\
(\text{let } ([X_b \ (o \ b_l \ b_r \ \dots \ b_m)]) \\
(\text{core-emit construct-e } b_m \\
\{(\text{expression } b) \ (\text{bindings } \emptyset)\}) \\
(\text{core-emit cont-rmv-e } b_m \ \emptyset) \\
X_b) \\
\text{subject to } \text{eval}[[X_b]] = \langle b, \emptyset \rangle$$

$$\text{inst-op}[\langle v, \langle v_r, \dots, o \rangle, \langle \langle N, \mathcal{E} \rangle, v_l, \dots \rangle \rangle] = \\
(\text{let } ([X_n \ (\text{begin} \\
(\text{core-emit construct-e } V \\
\{(\text{expression } N) \ (\text{bindings } \mathcal{E})\}) \\
(\text{core-emit cont-rmv-e } V \ \emptyset) \\
(\text{core-emit cont-add-e } V \\
\{(\text{continuation } \langle \text{op}, \langle v, v_r, \dots, o \rangle, \langle v_l, \dots \rangle \rangle)\}) \\
\text{inst-emit}[[N]])]) \\
\text{inst-op}[\text{eval}[[X_n]], \langle v, v_r, \dots, o \rangle, \langle v_l, \dots \rangle]) \\
\text{subject to } v = \langle V, \mathcal{E}_v \rangle$$

The tr-let and tr-begin represent the translation for the let and begin forms:

$$\text{tr-let}[(\text{let } ([X \ N]) \ M_1 \ \dots \ M_n)] = ((\lambda X \\
\text{tr-begin}[(\text{begin } M_1 \ \dots \ M_n)]) \ N)$$

$$\text{tr-begin}[(\text{begin } M)] = ((\lambda X \ X) \ M)$$

$$\text{tr-begin}[(\text{begin } M \ N \ \dots)] = ((\lambda X \ \text{tr-begin}[(\text{begin } N \ \dots)]) \ M)$$

To prove that the event instrumentation with inst and inst-emit is equivalent to the CESKT model, I go through each $\langle \langle M, \mathcal{E} \rangle, \Sigma, K, T \rangle$ machine state of the CESKT machine:

- **Case $M = (L N)$**

According to the [to-apply] reduction step, the CESKT machine generates

$$\langle \text{construct-e}, (L N), \text{core}, \{ (\text{expression } L) (\text{bindings } \mathcal{E}) \} \rangle = \text{evnt}_c$$

$$\langle \text{cont-add-e}, (L N), \text{core}, \{ (\text{continuation } \langle \text{ar}, \langle N, \mathcal{E} \rangle \rangle) \} \rangle = \text{evnt}_k$$

events.

By following the [inst5] rule, the instrumented version generates the same events before evaluating M .

- **Case $M = X$**

According to the [var] reduction step, the CESKT machine generates a

$$\langle \text{construct-e}, X, \text{core}, \{ (\text{expression } V) (\text{bindings } \mathcal{E}) \} \rangle = \text{evnt}_c$$

event where $\langle V, \mathcal{E} \rangle = \Sigma(\mathcal{E}(X))$.

By following the [inst1] rule, the instrumented version generates the same construct-e event.

- **Case $M = V$**

1. **Case $K = \langle \text{fn}, \langle (\lambda X N), \mathcal{E}_f \rangle, K \rangle$**

According to the [apply] reduction step, the CESKT machine generates

$$\langle \text{construct-e}, V, \text{core}, \{ (\text{expression } N) (\text{bindings } \mathcal{E}_f[X \leftarrow \sigma]) \} \rangle = \text{evnt}_c$$

$$\langle \text{function-e}, V, \text{core}, \{ (\text{name } X) (\text{slot } \sigma) (\text{value } \langle V, \mathcal{E} \rangle) \} \rangle = \text{evnt}_s$$

$$\langle \text{cont-rmv-e}, V, \text{core}, \emptyset \rangle = \text{evnt}_r$$

events.

In this case, M is a function, $(\lambda X N)$. By following the [inst2] rule, the environment variable is extended with a new entry, and the instrumented code will generate the same events.

2. **Case** $K = \langle \text{ar}, \langle N, \mathcal{E}_N \rangle, K \rangle$

According to the [apply-arg] reduction step, the CESKT machine generates

$$\langle \text{construct-e}, V, \text{core}, \{ (\text{expression } N) (\text{bindings } \mathcal{E}_N) \} \rangle = \text{evnt}_c$$

$$\langle \text{cont-rmv-e}, V, \text{core}, \emptyset \rangle = \text{evnt}_r$$

$$\langle \text{cont-add-e}, V, \text{core}, \{ (\text{continuation } \langle \text{fn}, \langle V, \mathcal{E} \rangle \rangle) \} \rangle = \text{evnt}_k$$

events.

By following the [inst4] rule, the instrumented version generates the same events before evaluating the N argument.

3. **Case** $K = \langle \text{op}, \langle v_{rest}, \dots, v_l, o \rangle, \langle \rangle, K \rangle$

According to the [prim] reduction step, the CESKT machine generates

$$\langle \text{construct-e}, V, \text{core}, \{ (\text{expression } b) (\text{bindings } \emptyset) \} \rangle = \text{evnt}_c$$

$$\langle \text{cont-rmv-e}, V, \text{core}, \emptyset \rangle = \text{evnt}_r$$

events.

By following the first clause of inst-op, the instrumented version generates the same events.

4. **Case** $K = \langle \text{op}, \langle v_R, \dots, o \rangle, \langle \langle N, \mathcal{E}_N \rangle, v_L, \dots \rangle, K \rangle$

According to the [prim-arg] reduction step, the CESKT machine generates

$$\langle \text{construct-e}, V, \text{core}, \{ (\text{expression } N) (\text{bindings } \mathcal{E}_N) \} \rangle = \text{evnt}_c$$

$$\langle \text{cont-rmv-e}, V, \text{core}, \emptyset \rangle = \text{evnt}_r$$

$$\langle \text{cont-add-e}, V, \text{core}, \{ (\text{continuation } \langle \text{op}, \langle \langle V, \mathcal{E} \rangle, v_R, \dots, o \rangle, \langle v_L, \dots \rangle \rangle) \} \rangle = \text{evnt}_k$$

events.

By following the second clause of inst-op, the instrumented version generates the same events.

5. **Case** $K = \langle \text{set}, \langle X, \mathcal{E} \rangle, K \rangle$

According to the [assign] reduction step, the CESKT machine generates

$$\langle \text{construct-e}, V, \text{core}, \{ (\text{expression } V_{old}) (\text{bindings } \mathcal{E}_{old}) \} \rangle = \text{evnt}_c$$

$$\langle \text{variable-update-e}, V, \text{core}, \{ (\text{name } X) (\text{slot } \sigma) (\text{value } \langle V, \mathcal{E} \rangle) \} \rangle = \text{evnt}_s$$

$$\langle \text{cont-rmv-e}, V, \text{core}, \emptyset \rangle = \text{evnt}_r$$

events.

By following the [inst6] rule, the instrumented version generates the same events.

- **Case $M = (o L N \dots)$**

According to the [to-prim] reduction step, the CESKT machine generates

$$\langle \text{construct-e}, (o L N \dots), \text{core}, \{ (\text{expression } L) (\text{bindings } \mathcal{E}) \} \rangle = \text{evnt}_c$$

$$\langle \text{cont-add-e}, (o L N \dots), \text{core}, \{ (\text{continuation } \langle \text{op}, \langle o \rangle, \langle \langle N, \mathcal{E} \rangle, \dots \rangle \rangle) \} \rangle = \text{evnt}_k$$

events.

By following the [inst5] rule, the instrumented version generates the same events.

- **Case $M = (\text{set } X N)$**

According to the [to-assign] reduction step, the CESKT machine generates

$$\langle \text{construct-e}, (\text{set } X N), \text{core}, \{ (\text{expression } N) (\text{bindings } \mathcal{E}) \} \rangle = \text{evnt}_c$$

$$\langle \text{cont-add-e}, (\text{set } X N), \text{core}, \{ (\text{continuation } \langle \text{set}, \langle X, \mathcal{E} \rangle \rangle) \} \rangle = \text{evnt}_k$$

events.

By following the [inst5] rule, the instrumented version generates the same events.

The core-emit form, therefore, is my basis of generating events for debugging. Since the core-emit form can simulate built-in events, and since built-in events can reconstruct the CESKT machine's entire state, core-emit provides all of the primitive debugging power needed.

4.3.1 Completeness of Model

If an implementation of the CESKT machine implements environments and program fragments with sharing, then each event logged to the trace component is a bounded increment in space consumption. Nevertheless, the machine's trace component contains enough information to reconstruct the rest of the machine state after each step. The ability to reconstruct the machine state provides evidence that the trace component contains all of the information that a debugger will need.

Theorem: For an input program, M , and a reduction sequence, $M = M_0 \mapsto M_1 \mapsto \dots \mapsto M_m = V$, in the CESK machine, the machine state after i reduction steps is $\langle\langle M_i, \mathcal{E}_i \rangle, \Sigma_i, K_i \rangle$ for $i \in [0, m]$. The following statement holds for a sequence of parallel reduction steps in the extended CESK machine:

Events generated in the parallel reduction steps can reconstruct the machine states in the CESK machine. At each reduction step, $M_i \mapsto M_j$, in the CESK machine, events generated by the parallel step, $M_i \mapsto M_j$, can reconstruct the next machine state, $\langle\langle M_j, \mathcal{E}_j \rangle, \Sigma_j, K_j \rangle$ from a transitioning state, $\langle\langle M_i, \mathcal{E}_i \rangle, \Sigma_i, K_i \rangle$.

Proof: Let $evnt_j \dots$ represent a sequence of events generated in a parallel reduction step, $M_i \mapsto M_j$. The `construct-e` events can help to reconstruct the expression and its enclosing environment, and `function-e`, `variable-update-e`, `cont-add-e`, and `cond-rmv-e` events can monitor the incremental changes of the store and continuation. In consequence, there exists a reconstruction metafunction that can map events to state changes.

The `reconstruct` metafunction that reconstructs the machine state is as follows:

$$\begin{aligned} \text{reconstruct}[\langle\langle M_i, \mathcal{E}_i \rangle, \Sigma_i, K_i \rangle, evnt_j \dots] &= \langle\langle M_j, \mathcal{E}_j \rangle, \Sigma_j, K_j \rangle \\ \text{subject to } evnt_j \dots &= evnt_c \ evnt_{rest} \dots, \\ evnt_c &= \langle \text{construct-e}, M_c, \text{core}, A_c \rangle, \\ A_c(\text{expression}) &= M_j, \\ A_c(\text{bindings}) &= \mathcal{E}_j, \\ \text{reconstruct-s}[\Sigma_i, \text{find-store-e}[evnt_{rest} \dots]] &= \Sigma_j, \\ \text{reconstruct-k}[K_i, evnt_{rest} \dots] &= K_j \end{aligned}$$

The sequence of generated events for each reduction step starts with a `construct-e` event, $evnt_c$, which allows reconstructing M_j and \mathcal{E}_j by using $A_c(\text{expression})$ and $A_c(\text{bindings})$ to find the `expression` and `bindings` attribute values in the A_c attribute table. The `find-store-e` metafunction searches through the rest of the events to find a store-related event or returns `false` on failure:

$$\begin{aligned}
\text{find-store-e}[\langle \rangle] &= \text{false} \\
\text{find-store-e}[\langle \text{store-e}, M_s, \text{core}, A_s \rangle \text{evnt}_{rest} \dots] &= \langle \text{store-e}, M_s, \text{core}, A_s \rangle \\
\text{find-store-e}[\text{evnt} \text{evnt}_{rest} \dots] &= \text{find-store-e}[\text{evnt}_{rest} \dots]
\end{aligned}$$

The `reconstruct-e` metafunction reconstructs the store for the next machine state:

$$\begin{aligned}
\text{reconstruct-s}[\Sigma, \text{false}] &= \Sigma \\
\text{reconstruct-s}[\Sigma, \langle \text{store-e}, M, \text{core}, A \rangle] &= \Sigma[\sigma \leftarrow v] \\
\text{subject to } A(\text{slot}) &= \sigma, A(\text{value}) = v
\end{aligned}$$

The `reconstruct-k` metafunction goes through the event sequence and reconstructs the continuation:

$$\begin{aligned}
\text{reconstruct-k}[K, \langle \rangle] &= K \\
\text{reconstruct-k}[K, \langle \text{cont-add-e}, M_k, \text{core}, A_k \rangle \text{evnt}_{rest} \dots] &= \text{reconstruct-k}[K_{new}, \text{evnt}_{rest} \dots] \\
\text{subject to assemble}[A_k(\text{continuation}), K] &= K_{new} \\
\text{reconstruct-k}[K, \langle \text{cont-rmv-e}, M_k, \text{core}, \emptyset \rangle \text{evnt}_{rest} \dots] &= \text{reconstruct-k}[K_{new}, \text{evnt}_{rest} \dots] \\
\text{subject to rmv-k}[K] &= K_{new} \\
\text{reconstruct-k}[K, \text{evnt} \text{evnt}_{rest} \dots] &= \text{reconstruct-k}[K, \text{evnt}_{rest} \dots]
\end{aligned}$$

where the `assemble` metafunction is

$$\begin{aligned}
\text{assemble}[\langle \text{fn}, \langle V, \mathcal{E} \rangle \rangle, K] &= \langle \text{fn}, \langle V, \mathcal{E} \rangle, K \rangle \\
\text{assemble}[\langle \text{ar}, \langle M, \mathcal{E} \rangle \rangle, K] &= \langle \text{ar}, \langle M, \mathcal{E} \rangle, K \rangle \\
\text{assemble}[\langle \text{op}, \langle \langle V, \mathcal{E} \rangle, \dots, o \rangle, \langle \langle M, \mathcal{E} \rangle, \dots \rangle \rangle, K] &= \langle \text{op}, \langle \langle V, \mathcal{E} \rangle, \dots, o \rangle, \langle \langle M, \mathcal{E} \rangle, \dots \rangle, K \rangle \\
\text{assemble}[\langle \text{set}, \langle X, \mathcal{E} \rangle \rangle, K] &= \langle \text{set}, \langle X, \mathcal{E} \rangle, K \rangle
\end{aligned}$$

and the `rmv-k` metafunction is

$$\begin{aligned}
\text{rmv-k}[\text{mt}] &= \text{mt} \\
\text{rmv-k}[\langle \text{fn}, \langle V, \mathcal{E} \rangle, K \rangle] &= K \\
\text{rmv-k}[\langle \text{ar}, \langle M, \mathcal{E} \rangle, K \rangle] &= K \\
\text{rmv-k}[\langle \text{op}, \langle \langle V, \mathcal{E} \rangle, \dots, o \rangle, \langle \langle M, \mathcal{E} \rangle, \dots \rangle, K \rangle] &= K \\
\text{rmv-k}[\langle \text{set}, \langle X, \mathcal{E} \rangle, K \rangle] &= K
\end{aligned}$$

With considerations of all possible cases of the M_i grammar and all possible reduction steps from M_i , the case-by-case proof is:

- **Case $M_i = (L N)$**

According to the [to-apply] reduction step, the extended CESK machine generates

$$\langle \text{construct-e}, (L N), \text{core}, \{ (\text{expression } L) (\text{bindings } \mathcal{E}_i) \} \rangle = \text{evnt}_c$$

$$\langle \text{cont-add-e}, (L N), \text{core}, \{ (\text{continuation } \langle \text{ar}, \langle N, \mathcal{E}_i \rangle \rangle) \} \rangle = \text{evnt}_k$$

events. Therefore,

$$\text{reconstruct}[\llbracket \langle \langle L N \rangle, \mathcal{E}_i \rangle, \Sigma_i, K_i \rangle, \text{evnt}_c \text{ evnt}_k \rrbracket] = \langle \langle L, \mathcal{E}_i \rangle, \Sigma_i, \langle \text{ar}, \langle N, \mathcal{E}_i \rangle, K_i \rangle \rangle$$

For the CESK machine, the reduction step for the M_i case is:

$$\langle \langle L N \rangle, \mathcal{E}_i \rangle, \Sigma_i, K_i \longrightarrow \langle \langle L, \mathcal{E}_i \rangle, \Sigma_i, \langle \text{ar}, \langle N, \mathcal{E}_i \rangle, K_i \rangle \rangle$$

where the next machine state is the same as the result of reconstruct.

- **Case $M_i = X$**

According to the [var] reduction step, the extended CESK machine generates a

$$\langle \text{construct-e}, X, \text{core}, \{ (\text{expression } V) (\text{bindings } \mathcal{E}) \} \rangle = \text{evnt}_c$$

event where $\langle V, \mathcal{E} \rangle = \Sigma_i(\mathcal{E}_i(X))$. Therefore,

$$\text{reconstruct}[\llbracket \langle \langle X, \mathcal{E}_i \rangle, \Sigma_i, K_i \rangle, \text{evnt}_c \rrbracket] = \langle \langle V, \mathcal{E} \rangle, \Sigma_i, K_i \rangle$$

which conforms to the CESK reduction rules.

- **Case $M_i = V$**

1. **Case $K_i = \langle \text{fn}, \langle (\lambda X N) \rangle, \mathcal{E}_f \rangle, K \rangle$**

According to the [apply] reduction step, the extended CESK machine generates

$$\langle \text{construct-e}, V, \text{core}, \{ (\text{expression } N) (\text{bindings } \mathcal{E}_f[X \leftarrow \sigma]) \} \rangle = \text{evnt}_c$$

$$\langle \text{function-e}, V, \text{core}, \{ (\text{name } X) (\text{slot } \sigma) (\text{value } \langle V, \mathcal{E}_i \rangle) \} \rangle = \text{evnt}_s$$

$$\langle \text{cont-rmv-e}, V, \text{core}, \emptyset \rangle = \text{evnt}_r$$

events. Therefore,

$$\text{reconstruct}[\llbracket \langle \langle M_i, \mathcal{E}_i \rangle, \Sigma_i, K_i \rangle, \text{evnt}_c \text{ evnt}_s \text{ evnt}_r \rrbracket] = \langle \langle N, \mathcal{E}_f[X \leftarrow \sigma] \rangle, \Sigma_i[\sigma \leftarrow \langle V, \mathcal{E}_i \rangle], K \rangle$$

which conforms to the CESK reduction rules.

2. **Case $K_i = \langle \text{ar}, \langle N, \mathcal{E}_N \rangle, K \rangle$**

According to the [apply-arg] reduction step, the extended CESK machine generates

$$\langle \text{construct-e}, V, \text{core}, \{ (\text{expression } N) (\text{bindings } \mathcal{E}_N) \} \rangle = \text{evnt}_c$$

$$\langle \text{cont-rmv-e}, V, \text{core}, \emptyset \rangle = \text{evnt}_r$$

$$\langle \text{cont-add-e}, V, \text{core}, \{ (\text{continuation } \langle \text{fn}, \langle V, \mathcal{E}_i \rangle \rangle) \} \rangle = \text{evnt}_k$$

events. Therefore,

$$\text{reconstruct}[\langle \langle M_i, \mathcal{E}_i \rangle, \Sigma_i, K_i \rangle, \text{evnt}_c \text{ evnt}_r \text{ evnt}_k] = \langle \langle N, \mathcal{E}_N \rangle, \Sigma_i, \langle \text{fn}, \langle V, \mathcal{E}_i \rangle, K \rangle \rangle$$

which conforms to the CESK reduction rules.

3. **Case** $K_i = \langle \text{op}, \langle v_{\text{rest}}, \dots, v_L, o \rangle, \langle \rangle, K \rangle$

According to the [prim] reduction step, the extended CESK machine generates

$$\langle \text{construct-e}, V, \text{core}, \{ (\text{expression } b) (\text{bindings } \emptyset) \} \rangle = \text{evnt}_c$$

$$\langle \text{cont-rmv-e}, V, \text{core}, \emptyset \rangle = \text{evnt}_r$$

events. Therefore,

$$\text{reconstruct}[\langle \langle M_i, \mathcal{E}_i \rangle, \Sigma_i, K_i \rangle, \text{evnt}_c \text{ evnt}_r] = \langle \langle b, \emptyset \rangle, \Sigma_i, K \rangle$$

which conforms to the CESK reduction rules.

4. **Case** $K_i = \langle \text{op}, \langle v_R, \dots, o \rangle, \langle \langle N, \mathcal{E}_N \rangle, v_L, \dots \rangle, K \rangle$

According to the [prim-arg] reduction step, the extended CESK machine generates

$$\langle \text{construct-e}, V, \text{core}, \{ (\text{expression } N) (\text{bindings } \mathcal{E}_N) \} \rangle = \text{evnt}_c$$

$$\langle \text{cont-rmv-e}, V, \text{core}, \emptyset \rangle = \text{evnt}_r$$

$$\langle \text{cont-add-e}, V, \text{core}, \{ (\text{continuation } \langle \text{op}, \langle \langle V, \mathcal{E}_i \rangle, v_R, \dots, o \rangle, \langle v_L, \dots \rangle \rangle) \} \rangle = \text{evnt}_k$$

events. Therefore,

$$\begin{aligned} \text{reconstruct}[\langle \langle M_i, \mathcal{E}_i \rangle, \Sigma_i, &= \langle \langle N, \mathcal{E}_N \rangle, \Sigma_i, \\ \langle \text{op}, \langle v_R, \dots, o \rangle, \langle \langle N, \mathcal{E}_N \rangle, v_L, \dots \rangle, K \rangle, &\langle \text{op}, \langle \langle V, \mathcal{E}_i \rangle, v_R, \dots, o \rangle, \\ \text{evnt}_c \text{ evnt}_r \text{ evnt}_k] &\langle v_L, \dots \rangle, K \rangle \end{aligned}$$

which conforms to the CESK reduction rules.

5. **Case** $K_i = \langle \text{set}, \langle X, \mathcal{E} \rangle, K \rangle$

According to the [assign] reduction step, the extended CESK machine generates

$$\langle \text{construct-e}, V, \text{core}, \{ (\text{expression } V_l) (\text{bindings } \mathcal{E}) \} \rangle = \text{evnt}_c$$

$$\langle \text{variable-update-e}, V, \text{core}, \{ (\text{name } X) (\text{slot } \sigma) (\text{value } \langle V, \mathcal{E}_i \rangle) \} \rangle = \text{evnt}_s$$

$$\langle \text{cont-rmv-e}, V, \text{core}, \emptyset \rangle = \text{evnt}_r$$

events. Therefore,

$$\text{reconstruct}[\langle \langle M_i, \mathcal{E}_i \rangle, \Sigma_i, K_i \rangle, \text{evnt}_c \text{ evnt}_s \text{ evnt}_r] = \langle \langle V_l, \mathcal{E} \rangle, \Sigma_i[\sigma \leftarrow \langle V, \mathcal{E}_i \rangle], K \rangle$$

which conforms to the CESK reduction rules.

- **Case** $M_i = (o \ L \ N \ \dots)$

According to the [to-prim] reduction step, the extended CESK machine generates

$$\langle \text{construct-e}, (o \ L \ N \ \dots), \text{core}, \{ (\text{expression } L) (\text{bindings } \mathcal{E}_i) \} \rangle = \text{evnt}_c$$

$$\langle \text{cont-add-e}, (o \ L \ N \ \dots), \text{core}, \{ (\text{continuation } \langle \text{op}, \langle o \rangle, \langle \langle N, \mathcal{E}_i \rangle, \dots \rangle \rangle) \} \rangle = \text{evnt}_k$$

events. Therefore,

$$\text{reconstruct}[\langle \langle (o \ L \ N \ \dots), \mathcal{E}_i \rangle, \Sigma_i, K_i \rangle, \text{evnt}_c \text{ evnt}_k] = \langle \langle L, \mathcal{E}_i \rangle, \Sigma_i, \langle \text{op}, \langle o \rangle, \langle \langle N, \mathcal{E}_i \rangle, \dots \rangle, K_i \rangle \rangle$$

which conforms to the CESK reduction rules.

- **Case** $M_i = (\text{set } X \ N)$

According to the [to-assign] reduction step, the extended CESK machine generates

$$\langle \text{construct-e}, (\text{set } X \ N), \text{core}, \{ (\text{expression } N) (\text{bindings } \mathcal{E}_i) \} \rangle = \text{evnt}_c$$

$$\langle \text{cont-add-e}, (\text{set } X \ N), \text{core}, \{ (\text{continuation } \langle \text{set}, \langle X, \mathcal{E}_i \rangle \rangle) \} \rangle = \text{evnt}_k$$

events. Therefore,

$$\text{reconstruct}[\langle\langle(\text{set } X \ N), \mathcal{E}_i\rangle, \Sigma_i, K_i\rangle, \text{evnt}_c \text{evnt}_k] = \langle\langle N, \mathcal{E}_i\rangle, \Sigma_i, \langle\text{set}, \langle X, \mathcal{E}_i\rangle, K_i\rangle\rangle$$

which conforms to the CESK reduction rules.

4.4 Mapping Events

Using `core-emit` directly to implement DSL events would be as painful as programming DSLs using a pure λ -calculus directly to implement the DSL's evaluation. The next step is to build a language for conveniently mapping events at one level of a language tower to events at the next level. That is, just as a module can implement a language layer by exporting macros that translate into the forms of a lower language level, a language-implementing module should export events that adapt the ones reported during evaluation in the lower language level. Besides translating lower-level events to a new level, the macros of a language-implementing module can inject fresh `emit` calls (which are ultimately translated into `core-emit` core forms), and the events generated by those `emit` forms are part of the language's event interface.

4.4.1 Declaring Events

In the same way that `define-syntax` binds an identifier to a macro, the `define-event` form binds an identifier to an event description:

```
(define-event event-pat composition-expr option ...)
```

```
composition-expr = event-pat
                  | (seq composition-expr ...)
                  | (disj composition-expr ...)
                  | (conj composition-expr ...)
                  | (rep composition-expr n)

event-pat = event-id
           | (event-id arg-id ...)

option = #:when expr
        | #:attributes ([id expr] ...)
        | #:specific
```

The *event-id* in a *event-pat* is a previously defined event, especially one from the language layers that the current module extends. For example, the `construct-e` event corresponds to the extension of the continuation to evaluate a subexpression, assuming that it is propagated from the core language to the current language layer.

The composition operators in a *composition-expr* are similar to the event expression operators in EBBA (Bates 1995) and involve the sequence operator, (*seq*), the disjunction operator, (*disj*), the conjunction operator, (*conj*), and the repetition operator, (*rep*). For example, an event defined as the disjunction of two other events causes the new event to be emitted whenever either of the other two is emitted.

The *options* of a *define-event* form further control the generation and content of an event. A *#:when* expression is evaluated each time the event might be generated, and the generation is blocked if the *#:when* expression's result is false. An *#:attributes* option adds symbol-keyed information to the event. The *#:specific* option connects an event declaration to the evaluation of an (*emit event-id*) form for the defined event.

An *expr* within *#:attributes* can access fields of the event matching *composition-expr* through an *attr* form to propagate or transform attributes values. When the event is declared with *#:specific*, then the expressions can additionally access variables that are in the environment of an associated (*emit event-id*) form.

To support parameterization over additional values for matching, an event can be defined as

```
(define-event (event-id arg-id ...) composition-expr option ...)
```

In that case, the *arg-ids* can be used in *option* expressions, and all references to *event-id* must have the form (*emit (event-id arg-expr ...)*).

An *emit* expression, which is typically generated by a macro, has either the form (*emit event*) or (*emit event syntax-expr*), where *event* either takes the form of *event-id* or (*event-id arg-expr*). When *syntax-expr* is included, it is used to associate program-source information with the event, and *syntax-expr* is typically a *#'* form that produces a template. An *emit* form with an associated *define-event* form to specify the shape of the emitted event's *event-id* can adjust the way the event is reported.

Only events declared with *define-event* or declared in a previous language and referenced with *emit* and then exported with *export-event* are part of a module's event interface:

```
(export-event event-id ...)
```

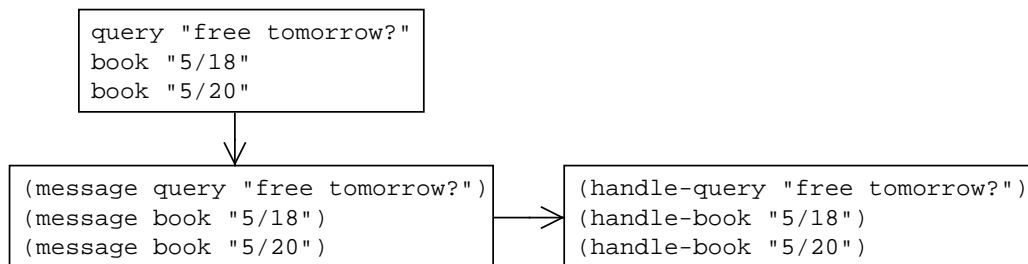
Events generated by lower language levels are not automatically propagated as events from the new language layer. In the unusual case that a module imports from different language modules,

the macros and events of all imported languages become visible to the importing module. More typically, however, a module imports a single language module, and so it sees only the syntactic forms (implemented by macros) and events of that language.

4.4.2 Examples and Kinds of Event Mappings

When performing event mapping from one language over another language, a variety of situations arise. Sometimes, events from the lower-level language can be propagated with small changes to the next layer. In other cases, events in a language correspond to a combination of events from a lower-level language and only when they happen in a particular evaluation context.

Suppose that a DSL program is expanded to the forms in the right box:



Each statement in the DSL program is parsed as a message S-expression, such as `book "5/20"` parsed as

```
(message book "5/20")
```

and the following macro specifies the message syntax expansion in terms of the lower-level language's `handle-query` and `handle-book` constructs:

```
(define-syntax (message stx)
  (syntax-case stx (query book)
    [(_ query m)
     #'(handle-query m)]
    [(_ book date)
     #'(handle-book date)]))
```

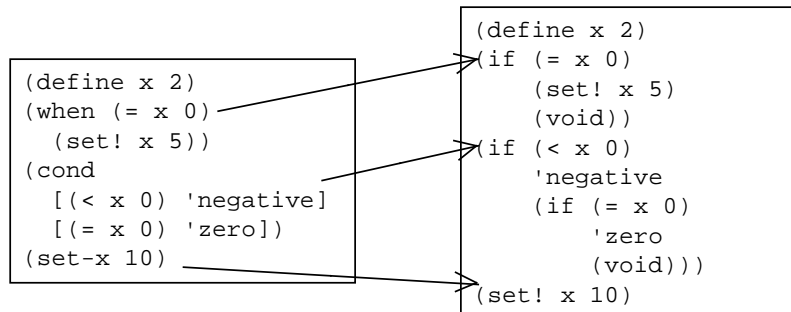
A message statement in the DSL translates to a function call in a procedural language. Suppose further that the procedural language generates a `subroutine-e` event when a defined function is called. To constrain an event consumption, `subroutine-e` is parameterized by the name of the called function.

If the execution of a `message` statement needs to be associated with a `message-e` event, the `message-e` can be defined in terms of events associated with `handle-query` and `handle-book` because of the fact that each statement is equal to the call of a `handle-query` or `handle-book` function:

```
(define-event message-e
  (disj (subroutine-e #:name 'handle-query)
        (subroutine-e #:name 'handle-book)))
```

The `message-e` falls into the category of a *generic event*, since the event can just be defined through `define-event` to specify the composition relationship of events.

As another example, suppose an L_2 language that just extends a lower-level L_1 language with conditional clauses, such as `when`, `cond`, and a new assignment syntax `set-x`. The L_2 program is on the left, and the L_1 program is on the right:



The related macros are:

```
(define-syntax (when stx)
  (syntax-case stx ()
    [(_ test then ...)
     #'(if ....)])) ; details omitted
(define-syntax (cond stx)
  (syntax-case stx ()
    [(_ (test then ...) rest ...)
     #'(if ....)])) ; details omitted
(define-syntax (set-x stx)
  (syntax-case stx ()
    [(_ v)
     #'(set! x v)]))
```

with `when` and `cond` expanded into `if` forms and `set-x` expanded into `set!`.

Because L_2 is an embedded language, without instrumentation of event mapping for L_2 , the

surface syntax in L_2 that belongs to L_1 should automatically have event support defined by L_1 . For example, suppose that `set!` is associated with a `variable-update-e` event in L_1 where the event is exported to L_2 . The execution of the `(set! x 5)` statement in L_2 will automatically generate a `variable-update-e` event, which is called a *host event*.

Even though `(set-x 10)` expands to `(set! x 10)`, the `variable-update-e` event associated with `set!` in L_1 will not be lifted to L_2 's event stream automatically. The event must be specifically propagated with an `emit` declaration. Avoiding automatic propagation helps hide internal implementation details. For example, a macro for a form that involves no explicit assignment might be transformed into a sequence that involves assignment,

```
....
(set! ....)
....
(set! ....)
....
```

but where the effects are local and not exposed. In the case of `set-x`, however, the effect is exposed and explicit; the `set-x` acts as a syntactic sugar over `set!` and shares the same semantics. Instrumenting the `set-x` macro with an explicit `emit` specification makes it part of the language's interface in the case of a `set-x` expansion.

```
(define-syntax (set-x stx)
  (syntax-case stx ()
    [(_ v)
     (with-syntax ([cur-stx stx])
       #'(begin
            (emit variable-update-e #'cur-stx)
            (set! x v))))))
```

This kind of `variable-update-e` event is called an *embedded event*.

Finally, suppose that L_2 's `cond` construct is associated with a `cond-e` event, which can be expressed as:

```
(define-event cond-e if-e)
```

This declaration registers a listener for the low-level `if-e` event and triggers `cond-e` generation upon `if-e` event generation. However, the code expansion contains several `if` forms from the `cond` form and an `if` form from the `when` macro, which would cause multiple `cond-e` event generation even if the DSL program just contains one `cond` form. In consequence, a different kind of event—

an *explicit event*—to specify run-time context is needed. The explicit event is declared with a `#:specific` option:

```
(define-event cond-e if-e #:specific)
```

and an `(emit cond-e)` form needs to be added to the macro transformation to register a listener for events associated with the evaluation of the expanded macro:

```
(define-syntax (cond stx)
  (syntax-case stx ()
    [(_ (test then ...) rest ...)
     (with-syntax ([cur-stx stx])
       #'(begin
            (emit cond-e #'cur-stx)
            (if ...)))))) ; details omitted
```

In general, because debugging events serve as back-end support for a debugger to expose run-time states and control program execution, event creation for a DSL is driven by debugger needs. For a DSL or any language, `define-event` can create an event based on existing, same-level events, and `emit` can map immediate, lower-level events. The cross-level event mapping falls into one of the above four categories of events: a generic event, a host event, an embedded event, and an explicit event.

4.4.3 Environment Information in Events

The `define-event` form supports capturing debugging information in event attributes either by extracting information from constituent events or by obtaining information at `emit` sites by directly using identifiers when the `#:specific` option is used if the former approach fails to capture needed information.

The `state-e` and `receive-msg-e` events illustrate the uses.

The `state-e` event declares a `state` attribute and obtains the debugging information from `construct-e`'s `bindings` attribute value:

```
(define-event state-e construct-e
  #:attributes ([state (attr construct-e 'bindings)]))
```

By declaring an option of `#:specific` in the `define-event` definition, the `receive-msg-e` event can access identifiers that are available in the environment of the associated `emit` form such as `m`, `message-tags`, `message-values`, and `message-time`.

```

(define-event receive-msg-e construct-e
  #:attributes ([message m]
                [type (last (message-tags m))]
                [values (message-values m)]
                [msg-time (message-time m)])
  #:specific)

(define-syntax (module-begin stx)
  (syntax-case stx ()
    [(_ decl ...)
     (with-syntax ([cur-stx stx])
       #'(base-module-begin
           ....
           (define handle-msg
             (lambda (m)
               (emit receive-msg-e #'cur-stx)
               (eval-msg m)))
           ....
           decl ...)))]))

```

In the CESKT model, `construct-e` captures all bindings in the environment at the point that the continuation is extended. Propagating all such bindings in an expanded program would reveal too many implementation details of expansion. For example, in Figure 4.3, the expansion of the `point` program into the procedure language introduces a new `point` identifier representing a function, which is irrelevant to the DSL program; a user of the DSL should see only that the program creates a `p` binding.

Consequently, the bindings created inside a macro expansion are not automatically collected by `construct-e`. To help programmers declare which bindings should be exposed for a given language layer, my system includes a `new-bindings` form that cooperates with `construct-e`. For the `point` example, I can add `new-bindings` to the previous `define-point` macro:

```

(define-syntax (define-point stx)
  (syntax-case stx (x y)
    [(_ name (x x-expr) (y y-expr) (op arg ...) ...)
     (with-syntax ([point-id (syntax-local-introduce #'point)])
       #'(begin
           (new-bindings name #'name)
           (define name (point-id x-expr y-expr))
           (send name op arg ...) ...)))]))

```

This `new-bindings` declaration causes the `construct-e` event emitted by the lower-level language to include the instantiation of `name` in its environment attribute.

4.4.4 Continuation Information in Events

Similar to environment information in emitted events, language implementations that expose continuation events need to control the emission of events to reflect the language's own continuation points, as opposed to the continuation points of the underlying language. The continuation events of the lower-level language will typically be too fine-grained and expose too much information. For example, using a simple `incr` form defined by

```
(define-syntax (incr stx)
  (syntax-case stx ()
    [(_ v amt)
     #'(set v (+ v amt))]))
```

an expression `(incr v 20)` expands into `(set v (+ v 20))`, which has three subexpressions that generate continuation frames, as opposed to the original expression's single subexpression.

To aid the construction of suitable continuation events, I provide `new-continuation` and `remove-continuation` forms, which expand to emit `cont-add-e` and `cont-rmv-e` events. The following is an example of specifying continuation events for the `incr` construct:

```
(define-syntax (incr stx)
  (syntax-case stx ()
    [(incr v amt)
     (with-syntax ([cur-stx stx])
       #'(begin0 (set v (begin (new-continuation "incr" #'cur-stx)
                               (+ v amt)))
                 (remove-continuation #'cur-stx)))]))
```

The `begin` and `begin0` forms both create sequences, but the `begin0` form returns the result of its first expression instead of its last expression. The `(new-continuation "incr" #'incr)` form emits a `cont-add-e` before the evaluation of `(+ v amt)`, and `(remove-continuation #'cur-stx)` emits a `cont-rmv-e` event afterward.

4.5 Run-Time Event Generation

The CESKT model with `core-emit` (Section 4.3) shows how the expanded version of a DSL program can generate core-level events, although it leaves abstract how the resulting event stream is consumed. The `define-event` form and associated constructs (Section 4.4), meanwhile, help language implementers specify filters and transformations of events to create a language-specific event interface. Run-time event generation and filtering ties these two pieces together. It consumes

the low-level events generated by the core language, and based on the specification of events at each language layer between the core and a DSL, it emits events that are suitable for consumption by a DSL-specific debugger.

Figure 4.7 illustrates the overall pipeline. A DSL implementation implies both a compiler from the DSL forms to core forms, an instrumentation of the resulting core forms to emit events, and a dependency graph of output events on core events. The events emitted at run time are triggered and transformed, based on the dependency graph, and a DSL debugger presents them to the user.

4.5.1 Event Dependency Construction

Figure 4.8 shows the representation of event-related structures as used by dependency construction. The dependency graph is represented by G with every t event node associated with its node-related information, $cmpt$. Each t event node tags an event with *layer* information, which distinguishes events generated at different language layers. Each `export-event` declaration modifies the *layer* tag to reflect each language layer.

The construction process starts with the bottom language and establishes event dependencies in a bottom-up fashion. For a primitive event definition or simple high-level event definition involving just one composition operator, a dependency is created for the event and each of its constituent events. For example, for

```
(define-event pe e1)
(define-event ce (disj e1 e2))
```

the event dependencies are as follows:

```
pe → e1
ce → e1
ce → e2
```

For event definitions involving nested event patterns in the composition expression, the dependency-construction process decomposes the event patterns by creating internal nodes for nested patterns in the dependency graph to simplify event recognition. For example, for

```
(define-event se (rep (disj e1
                           (conj e2 e3)
                           e4)
                      2))
```

two internal nodes, *ie1* and *ie2*, are constructed with their dependencies:

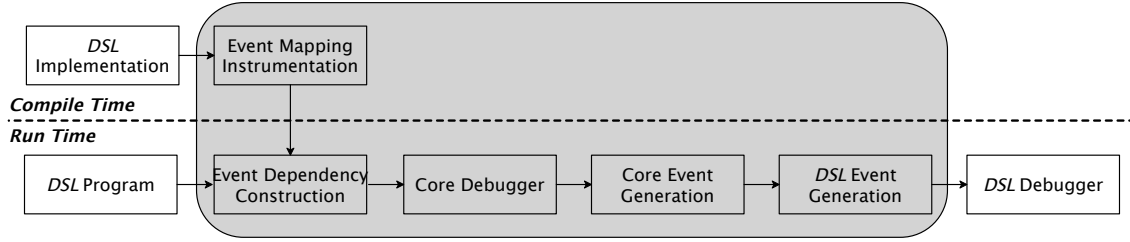


Figure 4.7. An event framework for domain-specific event creation and generation

$$\begin{aligned}
 \text{ename} &::= \text{etype} \mid (\text{etype } V \dots) & \text{state} &::= \text{evnt} \mid \mathbf{LIST}(\text{evnt}, \dots) \\
 \text{evnt} &::= \langle \text{ename}, \text{src}, \text{layer}, A \rangle & \text{mem} &::= \mathbf{LIST}(\text{ename}, \dots) \\
 t &::= \text{ename}@\text{layer} \\
 R &::= \{ (t \mathbf{LIST}(t, \dots)) \dots \} & G &::= \{ (t \text{ cmpt}) \dots \} \\
 F &::= \{ (t \text{ bool}) \dots \} & P &::= \{ (t \langle \text{bool}, A, \text{emit-src} \rangle) \dots \} \\
 C &::= \{ (t \langle \text{ctype}, S, \text{mem} \rangle) \dots \} & S &::= \{ (t \text{ state}) \dots \} \\
 D &::= \{ (t \langle \text{cond-exp}, \text{attr-exp}, \text{ntype} \rangle) \dots \} \\
 \text{cmpt} &::= \langle \text{cnv}, \langle \text{cond-exp}, \text{attr-exp}, \text{ntype} \rangle, \text{bool} \rangle \\
 \text{cnv} &::= t \mid \langle \text{ctype}, \mathbf{LIST}(t, \dots) \rangle \\
 \text{ctype} &::= \text{disj} \mid \text{conj} \mid \text{seq} \mid \langle \text{rep}, \text{ro} \rangle \\
 \text{ntype} &::= \text{internal} \mid \text{abstraction} \mid \text{false} \\
 \text{ro} &::= + \mid * \mid \text{integer} & \text{bool} &::= \text{true} \mid \text{false} \\
 \text{emit-src} &::= \text{srcloc} & \text{layer} &::= \text{a layer tag} \\
 \text{srcloc} &::= \langle \text{src}, \text{line}, \text{col} \rangle & \text{comp} &::= \text{a composition expression}
 \end{aligned}$$

Figure 4.8. Event-related structure representation

```

ie1 → (conj e2 e3)
ie2 → (disj e1 ie1 e4)
se → (rep ie2 2)

```

The event abstraction definition does not have any effect on the event dependency construction process, unless the definition is invoked in event patterns. For every e exported by `export-event`, if there exists no event dependency for e at this language layer, which means that the e event is inherited from the lower-level language, a new dependency connecting e to the lower-level e is created.

The semantics of `define-event` can be formulated as a register-dep metafunction, which adds an event dependency into the G graph:

$$\begin{aligned}
&\text{register-dep}[\![ename, comp, layer, cond-exp, attr-exp, bool, G]\!] = G[t \leftarrow cmpt] \\
&\text{subject to } ename@layer = t, \text{convert}[\![comp, layer]\!] = cnv, \\
&\quad \text{get-node-type}[\![ename]\!] = ntype, \\
&\quad \langle cnv, \langle cond-exp, attr-exp, ntype \rangle, bool \rangle = cmpt
\end{aligned}$$

where $comp$, $cond-exp$, $attr-exp$, and $bool$ are obtained from the event definition. The `convert` metafunction takes the *composition-expr* component of an event definition, $comp$, and the current layer, $layer$, and tags every event type in $comp$ with the correct layer information by determining the language layer to which the event type belongs. The $cond-exp$ and $attr-exp$ represent the expressions specified in an event definition's `#:when` and `#:attributes` options, and the $bool$ denotes if an event is an explicit event. The $ntype$ encodes the type of a node, whether it is an internal node created by the system or another kind of node.

4.5.2 DSL Event Generation

The preparation step for DSL event generation involves trimming event dependencies and building reverse references. To reduce the size of events, I keep only event dependencies needed by the top language level. As events originate from the core level, I need a push notification mechanism to trigger events at a higher language level. Every dependency, $e1 \rightarrow e2$, creates a reverse reference, $e2 \rightarrow e1$, and at run time, the *event processing unit* listens to core events and generates a DSL event according to the reverse mapping of event dependencies and the event definition's constraints.

Formally speaking, after the completion of G construction, the reverse references to event nodes, R , is constructed, and the F , C , and D tables store different aspects of node information. The C table

stores a mapping of a high-level tagged event to its recognition constraint and a recognition progress table, S .

The emission of core events initiates the DSL event generation process where the event processing unit looks up the parent nodes of an event in R , a list of event nodes to be triggered, and tries to trigger the parent node, t_p , one-by-one through `trigger`. Because the `emit` form affects the generation of an explicit event and an embedded event, a P table is used to record the evaluation of `(emit ename stx)` forms along with relevant event information:

$$\begin{aligned} \text{update-emit}[\![ename, A, stx]\!] &= P[t \leftarrow \langle \text{true}, A, \text{srcloc} \rangle] \\ \text{subject to } \text{get-layer-info}[\![stx]\!] &= \text{layer}, \text{ename@layer} = t, \\ \text{get-srcloc}[\![stx]\!] &= \text{srcloc} \end{aligned}$$

A is a set of attribute mappings associated with the $ename$ event. The trigger metafunction first checks if a t_p is an explicit event by looking up the F table and then attempts to generate an explicit event instance after the evaluation of `emit`. To illustrate, the case of `trigger` for a high-level, explicit t_p event is:

$$\begin{aligned} \text{trigger}[\![evnt, t_p, \text{src}_s, P, F, C, D]\!] &= \langle \text{ename}_p, \text{src}_s, \text{layer}_p, A_p \rangle \\ \text{subject to } t_p &= \text{ename}_p @ \text{layer}_p, \\ evnt &= \langle \text{ename}, \text{src}, \text{layer}, A \rangle, \\ F(\text{ename} @ \text{layer}) &= \text{true}, \\ P(\text{ename} @ \text{layer}) &= \langle \text{true}, A_{\text{emit}}, \text{srcloc} \rangle, \\ C(t_p) &= \langle \text{ctype}, S, \text{mem} \rangle, \text{updt-s}[\![S, evnt]\!] = S_{\text{new}}, \\ \text{check-comp-constraint}[\![S_{\text{new}}, \text{ctype}, \text{mem}]\!] &= \text{true}, \\ D(t_p) &= \langle \text{cond-exp}, \text{attr-exp}, \text{ntype} \rangle, \\ \text{get-comp-events}[\![C, D, t_p, \text{mem}]\!] &= \text{LIST}(\text{evnt}_c, \dots), \\ \text{check-cond-constraint}[\![\text{cond-exp}, \text{LIST}(\text{evnt}_c, \dots)]\!] &= \text{true}, \\ \text{get-attributes}[\![\text{attr-exp}, \text{LIST}(\text{evnt}_c, \dots), A_{\text{emit}}]\!] &= A_p \end{aligned}$$

The `check-comp-constraint` metafunction checks if the current recognition state meets its composition requirement on constituent events, and the `check-cond-constraint` metafunction checks if the *cond-exp* constraint is satisfied by obtaining its current matched constituent event values through `get-comp-events`. Since `emit` can either obtain attribute values from the `emit` evaluation context or from its event definition's constituent events, `get-attributes` returns the appropriate attribute values. The `get-attributes` metafunction also uses the `updt-attributes` metafunction to enrich the event with a `time` attribute and updates the `loc` attribute value if necessary.

The other cases of `trigger` generate a higher-level event dispatching on the t_p kind. If t_p is an embedded event, the generation rule is:

$$\begin{aligned}
&\text{trigger}[\![\text{evnt}, t_p, \text{src}_s, P, F, C, D]\!] = \langle \text{ename}_p, \text{src}_p, \text{layer}_p, A_p \rangle \\
&\text{subject to } t_p = \text{ename}_p @ \text{layer}_p, \\
&\quad \text{evnt} = \langle \text{ename}, \text{src}, \text{layer}, A \rangle, \\
&\quad P(\text{ename} @ \text{layer}) = \langle \text{true}, A_{\text{emit}}, \text{srcloc} \rangle, \\
&\quad \text{get-source}[\![\text{srcloc}, \text{src}]\!] = \text{src}_p, \\
&\quad \text{updt-attributes}[\![A, \text{srcloc}]\!] = A_p
\end{aligned}$$

Depending on the value of srcloc , get-source chooses a source value between srcloc and src .

If t_p is a primitive, generic event and the run-time condition of the event is satisfied, a new event instance is generated by packing appropriate attribute values from the event it depends on:

$$\begin{aligned}
&\text{trigger}[\![\text{evnt}, t_p, \text{src}_s, P, F, C, D]\!] = \langle \text{ename}_p, \text{src}, \text{layer}_p, A_p \rangle \\
&\text{subject to } t_p = \text{ename}_p @ \text{layer}_p, \\
&\quad \text{evnt} = \langle \text{ename}, \text{src}, \text{layer}, A \rangle, C(t_p) = \text{NONE}, \\
&\quad D(t_p) = \langle \text{cond-exp}, \text{attr-exp}, \text{ntype} \rangle, \\
&\quad \text{check-cond-constraint}[\![\text{cond-exp}, \text{evnt}]\!] = \text{true}, \\
&\quad \text{get-attributes}[\![\text{attr-exp}, \text{evnt}]\!] = A_p
\end{aligned}$$

Otherwise, if t_p is a high-level, generic event, the event generation rule is similar to the high-level, explicit event case but with a different mechanism for generating A_p :

$$\begin{aligned}
&\text{trigger}[\![\text{evnt}, t_p, \text{src}_s, P, F, C, D]\!] = \langle \text{ename}_p, \text{src}_s, \text{layer}_p, A_p \rangle \\
&\text{subject to } t_p = \text{ename}_p @ \text{layer}_p, C(t_p) = \langle \text{ctype}, S, \text{mem} \rangle, \\
&\quad \text{updt-s}[\![S, \text{evnt}]\!] = S_{\text{new}}, \\
&\quad \text{check-comp-constraint}[\![S_{\text{new}}, \text{ctype}, \text{mem}]\!] = \text{true}, \\
&\quad D(t_p) = \langle \text{cond-exp}, \text{attr-exp}, \text{ntype} \rangle, \\
&\quad \text{get-comp-events}[\![C, D, t_p, \text{mem}]\!] = \text{LIST}(\text{evnt}_c, \dots), \\
&\quad \text{check-cond-constraint}[\![\text{cond-exp}, \text{LIST}(\text{evnt}_c, \dots)]\!] = \text{true}, \\
&\quad \text{get-attributes}[\![\text{attr-exp}, \text{LIST}(\text{evnt}_c, \dots)]\!] = A_p
\end{aligned}$$

In the process of upward event generation, the trigger metafunction updates events with appropriate src information so that events belonging to different language layers can be differentiated. In the end, the event processing unit just keeps the event instances at the DSL level and directs these events to event handlers set up for a debugger implementation.

CHAPTER 5

DEBUGGING FRAMEWORK

The approach of mapping DSLs to domain-specific events brings many benefits to DSLs, and domain-specific events can enable a debugging framework that supports building effective, domain-specific debuggers with a low cost. A DSL-construction tool usually supports creating a variety of DSLs that differ in program states and evaluation models, but a considerable amount of debugger effort can be reused to facilitate the collection and presentation of debugging information. I propose a debugging framework that contains a *back end* providing support for data collection of debuggers and a *front end* incorporating a collection of reusable, DSL-friendly debugging views and other primitives of interface implementation. Since the event work in Chapter 4 enables encapsulating debugging information in events, the back-end design is based on the previous event mechanism in Chapter 4, and this chapter presents the design for the front end and an evaluation of a debugging framework I implemented on the Racket platform, Ripple.

5.1 Front End

Presenting debugging information in the front end in a user-friendly way is a difficult task requiring work in interface design and implementation. My front-end design aims to incorporate a suite of tools to reduce the cost of interface development and employs a composable approach to enable flexible debugger extensions. The front end involves support for three debugging modes where a different category of interface tools is built for each debugging mode.

5.1.1 Elementary Mode

Because graphical operations offer an intuitive, user-friendly control to interact with debuggers, the elementary mode focuses on graphical control where debugger developers need to implement a set of graphical views for a debugger.

For the interface layout, I use a concept of a *view* to represent a graphical widget with a designated functionality. A view denotes a user interface composition unit, which enables customization

of a debugger interface to cater to domain needs. For example, the interface can have a source view that is responsible for displaying the source program and providing program-related graphical operations such as setting breakpoints and highlighting code segments.

The framework provides an *interface-form* to assist the interface construction:

```

interface-form ::= (ui 'elementary layout-form)
layout-form ::= name
                | (h-layout layout-form ... b ...)
                | (v-layout layout-form ... b ...)
b ::= number
name ::= a token such as variable-view

```

For example, the following `ui` form specifies a layout in the elementary mode:

```

(ui
 'elementary
 (h-layout (v-layout step-view source-view 1/6 5/6)
            variable-view
            3/5 2/5))

```

The first argument to `ui` instantiates a debugging mode. The `h-layout` and `v-layout` operators create horizontal layouts and vertical layouts, respectively, for constituent windows having specified geometric space percentages.

After specification of the interface layout, a view implementation is needed to enable graphical operations. Because many view implementations can be reused across different debuggers, my framework provide a library of reusable views:

- A source view supports displaying a program in a text editor, highlighting code, and setting breakpoints.
- A variable view supports displaying variable bindings.
- A step view supports event-based execution control.
- A continuation view supports visualizing control flow information.
- An event property view supports displaying event attribute information.
- An event view supports visualizing events in timelines.

One of the difficulties of writing a debugger involves information visualization. Debugger developers may know what domain concepts to present to users, but how to present the information

remains a challenge. By keeping the system views as a reservoir of standard solutions to common problems, developers can simply adopt the views as solutions or use the views as sources of inspiration for new solutions. In the above ui declaration, the `step-view`, `source-view`, and `variable-view` names refer to the built-in step view, source view, and variable view.

Figure 5.1 shows the process of constructing a debugger in the elementary mode, which involves:

- starting from the front end to decide the features and layout of a DSL debugger,
- instrumenting the DSL implementation with events in the back end, and
- writing event handlers to connect events with the front-end interface and implementing debugger interface features.

To assist the specification of event handlers and implementation of interface views that are beyond the built-in views, the framework provides the following primitives:

```

handler-form ::= (on ename expr ...)
view-form ::= (view 'name')
event-form ::= (current ename) | (traces ename)
               | (attr ename 'name') | (step)

```

The *handler-form* registers an event handler, and *view-form* obtains the panel in the interface that holds the view named *name*. The `current` primitive obtains the newest event instance, and `traces` obtains a history of event instances. The `attr` primitive helps to access an event's attribute value, and `(step)` provides an execution-control facility, which works with an `on` from to support pausing execution whenever an event happens. Figure 4.8 contains the *ename* definition.

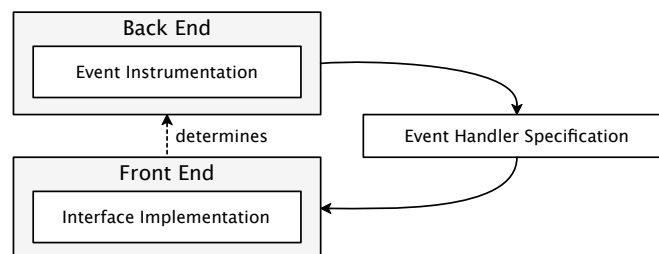


Figure 5.1. Debugger construction process

To illustrate, suppose that a `custom-e` event needs to be connected with the front-end view, which is named as `custom-view` in a ui layout specification. Whenever `custom-e` happens, the custom view needs to update its content. The corresponding configuration can be written as:

```
(define custom-view-widget%
  (class object%
    data ....
    operations ....))

(define custom-wgt
  (new custom-view-widget% [panel (view 'custom-view)]))

(on custom-e
  (let ([e (current custom-e)])
    (send custom-wgt update-content (attr e 'content))
    (step))))
```

First, a new widget class with custom data and operations is implemented, and a widget object, `custom-wgt`, is created where `(view 'custom-view)` accesses the graphical panel reserved for the view. The `on` form sets up an event handler for the `custom-e` event, which is defined in the back end. The event handler specifies that whenever a new instance of `custom-e` happens, the `update-content` operation of the custom view is called with the information obtained from the event's `content` attribute. The execution of `(step)` pauses the program's execution, and the program resumes execution when a *Go* or *Step* button is clicked in the step view.

5.1.2 Motivation for Other Possible Modes

Features in the elementary mode are useful for debugging programs graphically, but fixed operations cannot anticipate all needs because the efficiency of debugging depends on source program complexities. For example, a DSL program can deal with messages spanning a significant period of time, and a view displaying the message history in the elementary mode would become flooded, making graphical extraction difficult.

Moreover, DSLs are no longer limited to end users who have limited programming skills, which creates a need for flexible debugger extensions. For example, LaTeX and SQL are widely used by traditional programmers to typeset documents and perform database queries, respectively. Because of their expressive power in solving problems in particular domains, DSLs are also popular among traditional GPL programmers. Therefore, by taking into account the possible diversity of users' programming backgrounds ranging from programming proficiency to limited programming knowl-

edge, I present two more debugging modes: an intermediate mode and an advanced mode. The interface in the elementary mode focuses on purely graphical control, but advancing the interface mode increases programmatic control support.

5.1.3 Intermediate Mode

The intermediate mode, which offers debugger extensions where users don't "have to know everything to do anything" but can learn debugging incrementally, acts as a bridge between the elementary mode and the advanced mode.

The intermediate features consist of graphical assistance and simple programmatic control. For the graphical assistance, I include features that are relatively easy for end users to learn and use such as event inspection, visualization, and assertions (inspired by the positive effect of assertions on end users for the spreadsheet paradigm (Burnett et al. 2003)). End users being able to construct some useful visualization for a problem such as visualizing numeric values in a line is desirable, but this mode just focuses on visualizations for events. The intermediate mode focuses on automatic support, and the burden of constructing domain-specific visualizations is shifted to the elementary mode since debugger developers can construct appropriate visualizations. Since end users also have the ability to learn and use a programming language containing high-level primitives (Nardi 1993), for programmatic control, I restrict accessible programming primitives to be a small set of event-oriented primitives, which allows users to experiment with simple scripting over debugging events and eventually transition to the advanced mode.

The primitives available in the intermediate mode are:

```

event-def ::= (define-event ename comp option ...)
            | (define-event (ename X ...)
              comp option ...)
event-expr ::= (on ename expr ...)
            | (current ename)
            | (traces ename)
            | (attr ename 'name)
            | (log e)
            | (timeline ename ...)
            | (assert-exists ename)
            | (assert expr)
e ::= an event instance

```

The `define-event` primitive allows defining a new event. The `on` primitive allows writing an event handler specifying actions to do when an event with an *ename* is triggered. The `current` primitive obtains the latest event instance, and `traces` obtains the history of an event. The

`attr` primitive obtains an *ename* event's attribute value. The `log` primitive prints an event, and `timeline` visualizes an event in a timeline. The `assert-exists` primitive checks that an event exists, and `assert` checks a condition represented by *expr* to be true.

5.1.4 Advanced Mode

The advanced mode gives debugger users complete, programmatic control over writing a debugger with desired debugging operations. The GPL community considers a dedicated debugging language allowing scripting of problem-specific debugging commands to be the most powerful approach to effective debugging (Johnson 1977; Marceau et al. 2006; Olsson et al. 1990; Winterbottom 1994). Inspired by the advantages of a dedicated debugging language for GPLs, the advanced mode should provide a debugging language.

For the debugging language, I choose an extension of a GPL with the primitives available in the intermediate mode for the following reasons. First, since the execution details of a DSL are captured by domain-specific events defined in the back end, the domain-specific events are accessible in the language, and users can manipulate events by using event primitives. Second, a GPL affords computational power to achieve any computable debugging task and requires little transition overhead for a traditional programmer to use the advanced mode.

The debugging language opens the door to unlimited operations, and learning a debugging language is more useful and rewarding on DSL platforms than on GPL platforms. A GPL platform debugger typically works for one GPL, and a new syntax for a debugging language is usually introduced for each GPL platform debugger. However, for the DSL platform debugger, a single debugging language can be used universally for different DSLs because many DSLs are created on the same platform.

5.1.5 Front-End Construction

By considering the background and needs of potential users, a DSL debugger can have more than one debugging mode. The `(ui 'elementary layout-form)` form creates an elementary mode requiring the most interface design and implementation effort to create an effective debugger. However, a debugging framework can provide automatic support for the intermediate mode and the advanced mode. In Ripple, with just declarations of `(ui 'intermediate)` and `(ui 'advanced)`, the intermediate and advanced features are made available to users.

CHAPTER 6

APPLICATIONS

To validate the design of the proposed debugging framework, I implemented a debugging framework, Ripple, on Racket. I worked with three DSLs, and I will demonstrate the domain-specific debuggers built with Ripple in this chapter.

6.1 Scratchy Debugger

Scratchy¹ is an imperative language to write Scratch-like applications. Figure 6.1 shows the interface of the Scratchy debugger, which consists of two tabs: an elementary tab and an intermediate tab. Each tab represents a *debugging mode* with designated programming complexity.

In the elementary tab, clicking the *Debug* button in the top-right corner activates the debugger configuration for the elementary mode. This sets up event listeners, enables graphical operations for built-in views, and generates custom views built by developers. Figure 6.2 shows a debugging scenario in the *elementary mode*. The interface contains a step view in the top-left pane, a source view in the bottom-left pane, and a variable view on the right. The *source view* contains the Scratchy source code, and the *step view* controls the execution of the program. For example, the *Step* button in the step view supports executing the program to the next break event. The *Go* button supports executing the program to the end, and the *Pause* button supports pausing the program execution. A *break event* can be created by reaching a breakpoint during program execution or by executing a statement during stepping. When the mouse is hovering over different positions in the source code, breakable points (usually the beginning of a statement) are marked by circles, and a *breakpoint* can be created by right-clicking a breakable point. At each step, the *variable view* displays the current program state in terms of variable bindings.

In Figure 6.2, the execution stops at the `turn by random 5` statement, and the variable view displays the current variable bindings in the program such as `rightWall` and `swimmer`. Each

¹<https://docs.racket-lang.org/scratchy>

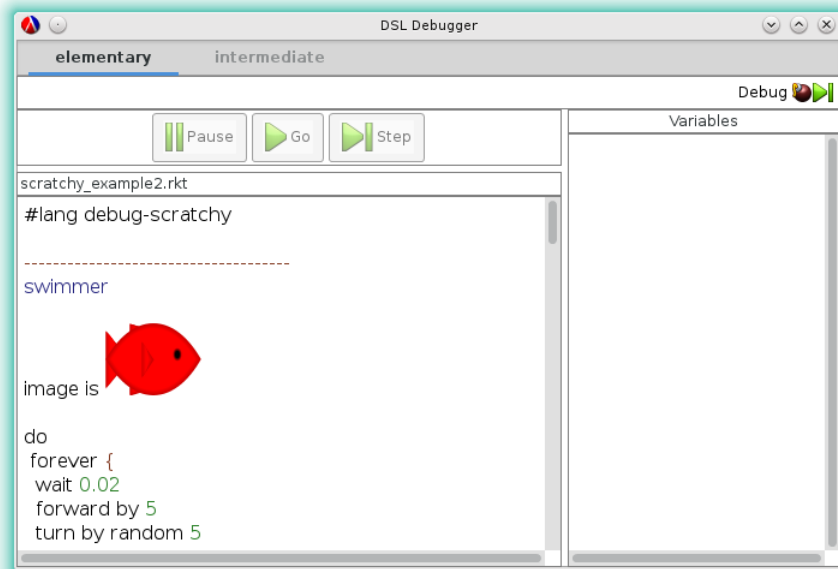


Figure 6.1. Scratchy debugger

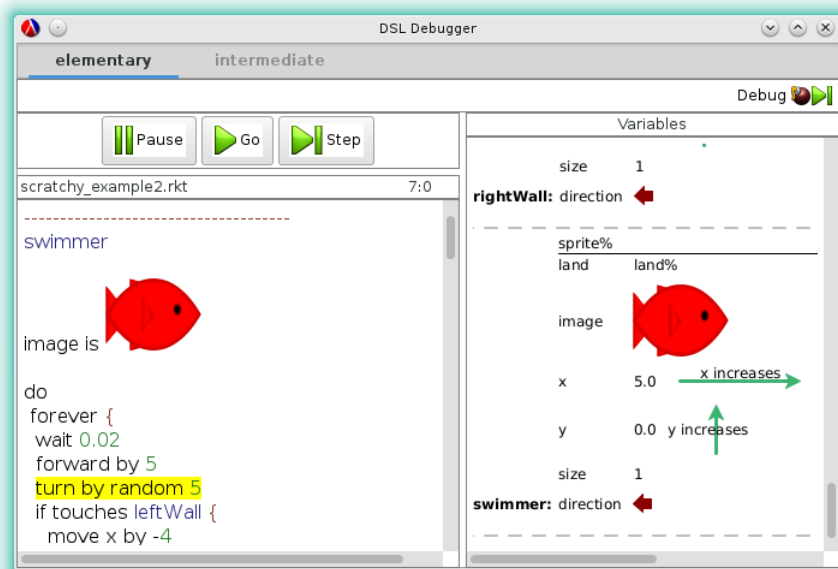


Figure 6.2. Scratchy debugger interface in an elementary mode

variable binding consists of a variable name and a variable value. If a variable is a sprite object such as the swimmer, the sprite's properties including *land*, *image*, *x*, *y*, *size*, and *direction* are shown, and visual annotations are added to better understand values. For example, instead of displaying the *direction* property of *swimmer* to be a numerical 270 value, the value is rendered as an arrow pointing in the direction that the sprite is turning to. The arrow annotation of “x increases” and “y increases” with the *x* and *y* properties also illustrates that the *x* value and *y* value increases when the sprite moves towards the right or top, respectively.

By clicking the intermediate tab, the debugger switches to the intermediate mode. Unlike the elementary mode that provides fixed operations, the *intermediate mode* provides selective control for problems that require different combinations of debugging operations or information processing. There are two kinds of assistance for debugging: event-based, graphical control and simple, programmatic control.

Figure 6.3 demonstrates a use of the graphical control. Yellow circles in the source program mark the existence of debugging events for possible exploration. Depending on debugging needs, a user right-clicks a yellow circle, and the program displays a list of available events in a pop-up menu. By choosing the event that is relevant to the debugging task, a second-level menu shows a list of event-related operations:

- Inspect this event: Prints the value of this event in the output view, which can be opened by clicking the *Output* button.
- Step this event: Pauses the execution whenever this event is fired and pops up a window that shows the event information and provides a *Continue* button to resume execution.
- Visualize this event: Visualizes this event in a timeline, which can be viewed by clicking the *Show Timeline* button in the output view.
- Assert event existence: Reports an error if a specified event is never triggered before the end of execution. When the program finishes execution, clicking the *Check Event Existence* button will check the existence of all specified events.
- Assert event attributes: Pops up a graphical window (Figure 6.4) allowing specifying data constraints of event attributes and reports an error if the constraints are not satisfied.

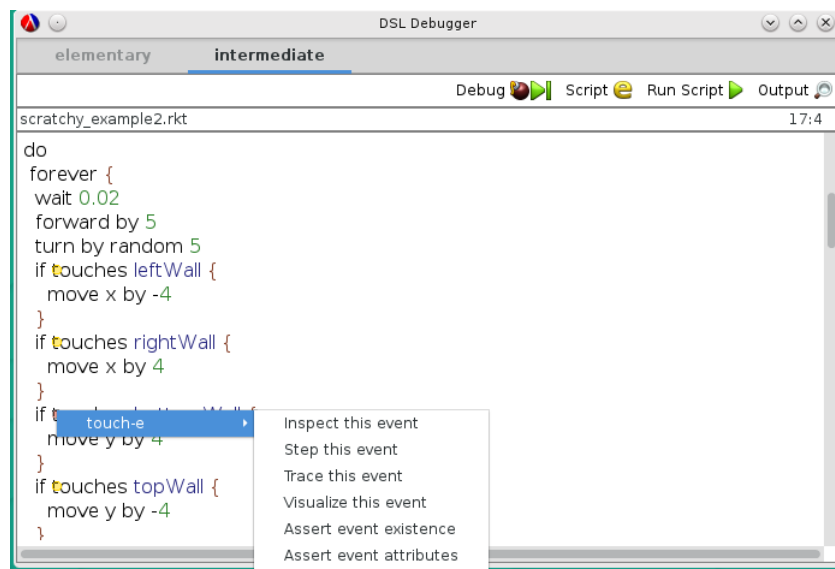


Figure 6.3. Scratchy debugger interface in an intermediate mode

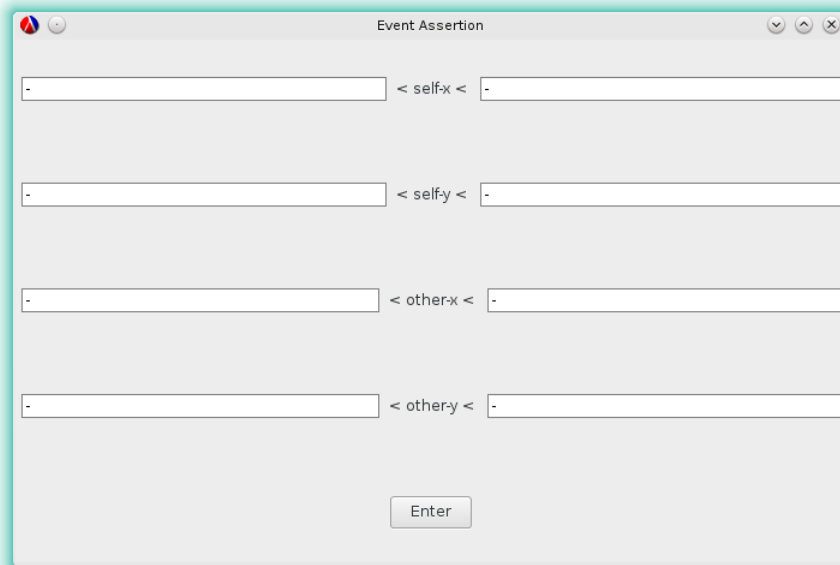


Figure 6.4. Event assertion

Clicking the *Script* button in the intermediate interface brings a script view next to the source view. Users who have programming background can use the *script view* to write scripts to process and analyze debugging data. The `construct-e` and `touch-e` events are fundamental events provided by the event instrumentation for event processing in the intermediate mode. A `construct-e` event is fired whenever a statement is entered, and a `touch-e` event is fired whenever a `touches` statement is executed. Each kind of event carries event-specific debugging information in attribute values. The `construct-e` event carries a `bindings` attribute, and the `touch-e` event has `self-sprite`, `self-x`, `self-y`, `other-sprite`, `other-x`, and `other-y` attributes with debugging information about two sprites that collide with each other.

6.1.1 Finding Scratchy Bugs in the Elementary Mode

The “`scratchy_example2.rkt`” program shown in Figure 6.5 contains a bug. The execution opened a graphical window that showed a swimming fish and a blue rectangular pool. The fish was expected to swim within the pool, but somehow the fish escaped the pool.






Since the fish was able to leave the pool through the pool’s right wall, breakpoints can be set at areas of code that are related to the `rightWall` to check the state of the fish. Therefore, a breakpoint is set by right-clicking the beginning of `move x by 4` inside the `if touches rightWall` block. The program executed to the breakpoint with a click of the *Step* button, and the variable view was updated. As shown by Figure 6.6, the `x` property of the fish is about 116.63. With another click of the *Step* button, the `x` property is increased to 122.97 (Figure 6.7).

The “`x increases`” annotation beside the `x` value indicates that when the fish moves in a right direction horizontally, the `x` value will increase. In this debugging scenario, the fish’s `x` value kept increasing after touching the pool’s right wall, which deviated from the desired behavior. If the fish touches the right wall, the fish should change its horizontal movement to the left, which means that the `x` value should decrease. Therefore, the code inside the `if touches rightWall` block should be `move x by -4`. Because the `if touches leftWall` block is an opposite case of `if touches rightWall`, the code for the `leftWall` case should be changed to be `move x by 4`, too.

6.1.2 Finding Scratchy Bugs in the Intermediate Mode

The execution of the “`scratchy_example3.rkt`” shown in Figure 6.8 produced an error again with the fish getting out of the pool.

```

-----
swimmer
image is 
do
  forever {
    wait 0.02
    forward by 5
    turn by random 5
    if touches leftWall {
      move x by -4
    }
    if touches rightWall {
      move x by 4
    }
    if touches bottomWall {
      move y by 4
    }
    if touches topWall {
      move y by -4
    }
  }
}
-----
rightWall
image is 
x is 150
-----
leftWall
image is 
x is -150
-----
topWall
image is 
y is 100
-----
bottomWall
image is 
y is -100

```

Figure 6.5. A buggy Scratchy program

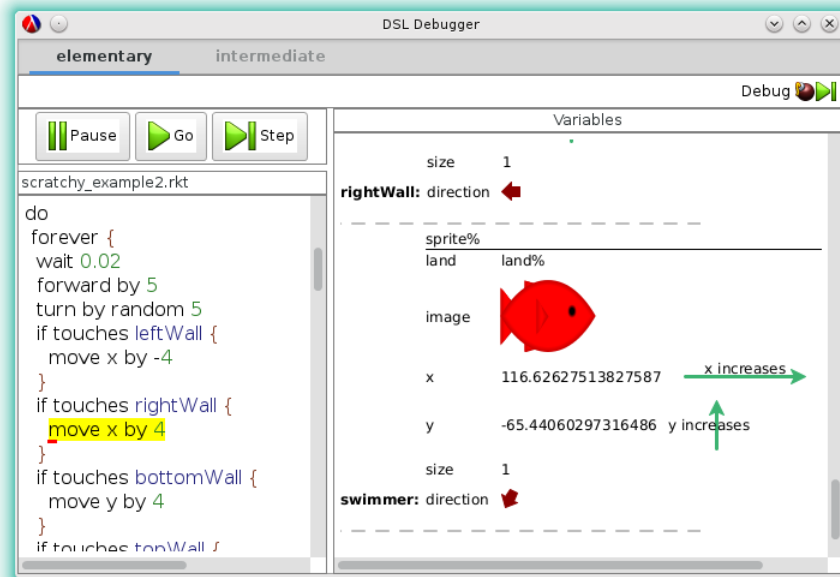


Figure 6.6. Scratchy debugger interface after first step

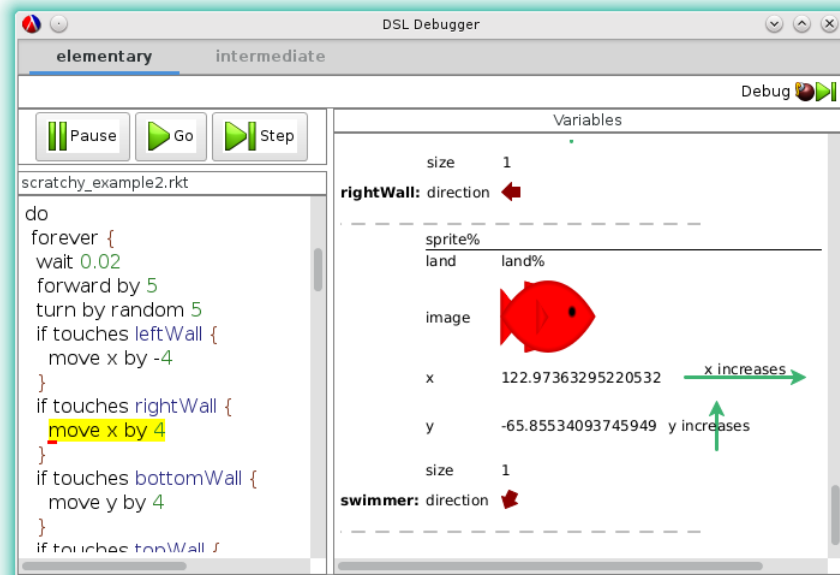







Figure 6.7. Scratchy debugger interface after second step

```

-----
swimmer
image is 
do
  forever {
    wait 0.02
    forward by 2
    turn by random 5
    if touches leftWall {
      move x by -2
    }
    if touches rightWall {
      move x by 2
    }
    if touches bottomWall {
      move y by 2
    }
    if touches topWall {
      move y by -2
    }
  }
}
-----
rightWall
image is 
x is 150
-----
leftWall
image is 
x is -150
-----
topWall
image is 
y is -100
-----
bottomWall
image is 
y is 100

```

Figure 6.8. Scratchy example

To find the cause of this erroneous behavior, a programmer used the elementary mode to debug the program first but ran four trials to find the problem. Because the behavior of the fish is specified in the `forever` loop, which executes a block of code endlessly, detecting erroneous information by stepping through the code without a breakpoint in the first trial was time-consuming. In the second trial, the programmer set a breakpoint at the `move y by 2` statement inside the `if touches bottomWall` block because the erroneous behavior involved the fish touching the bottom wall. However, the debugger was never able to reach the statement, and in the third trial, a new breakpoint was set at the `move y by -2` statement inside the `if touches topWall` block. The execution reached the breakpoint set at the `move y by -2` statement, but because the fish never touched the top wall in the generated window, the reason for the execution of this breakpoint was not clear. In the last trial, the debugger was run again with the two previous breakpoints and the execution still reached the breakpoint inside the `if touches topWall` block instead of the `if touches bottomWall` block. The programmer went through the variables in the variable view to check the program states and finally found the problem: incorrect initializations for the top wall and the bottom wall. The variable view showed that the `topWall`'s `y` value is -100 and that the `bottomWall`'s `y` value is 100, but their values should be switched.

Because the elementary mode uses a stepping-based approach, it suffers from the shortcomings of a stepping-based approach, which involve difficulty of setting appropriate breakpoints and difficulty of extracting relevant information from a fixed interface. Since the intermediate mode offers selective control, the intermediate mode was used to debug the same program. Choosing “Step this event” for the `touch-e` event at four `touches` places turned four circles red where a circle represents an event (Figure 6.9). Clicking the *Debug* button brought up an *Event Info* window showing the debugging information about the `touch-e` event. The *Continue* button in the pop-up window enabled execution to the `touches bottomWall` expression. Figure 6.10 shows the event information about the current `touch-e` event where `self-sprite` and `other-sprite` show the sprite values of two colliding sprite objects, and their `x` and `y` values are represented by `self-x` and `self-y`, and `other-x` and `other-y`, respectively. In the figure, `self-sprite` represents the fish, and `other-sprite` represents the bottom wall. Variables of `other-x` and `other-y` show that `x` and `y` values of the bottom wall are 0 and 100. By clicking the *Continue* button again, the execution paused at `touches topWall` as shown in Figure 6.11. Figure 6.11 shows that the `x` and `y` values of the top wall are 0 and -100, where the `y` value seemed to be incorrect. If the bottom wall's `y` value

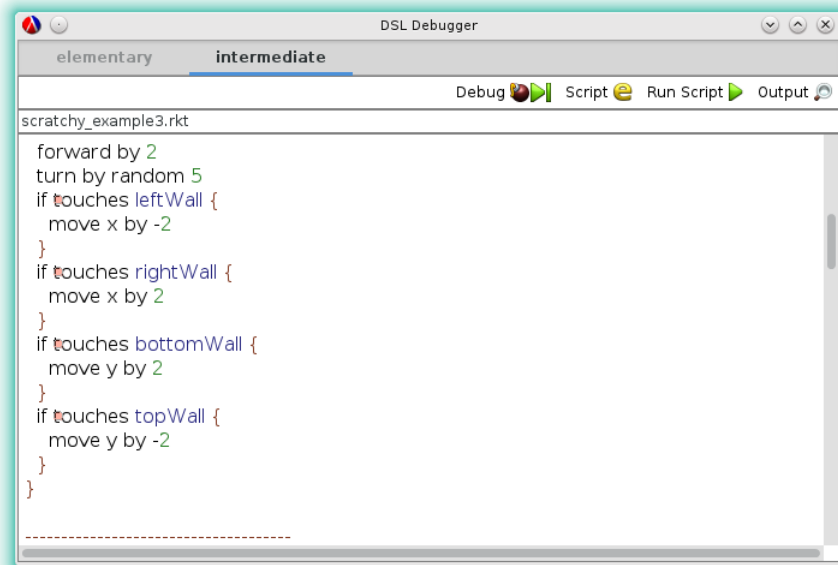


Figure 6.9. Enabling stepping in the intermediate mode

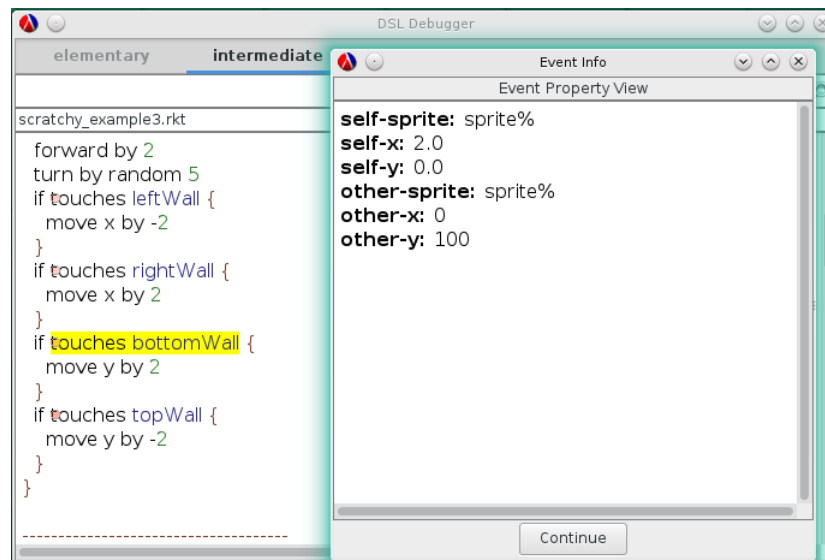


Figure 6.10. Stepping to the bottom wall statement

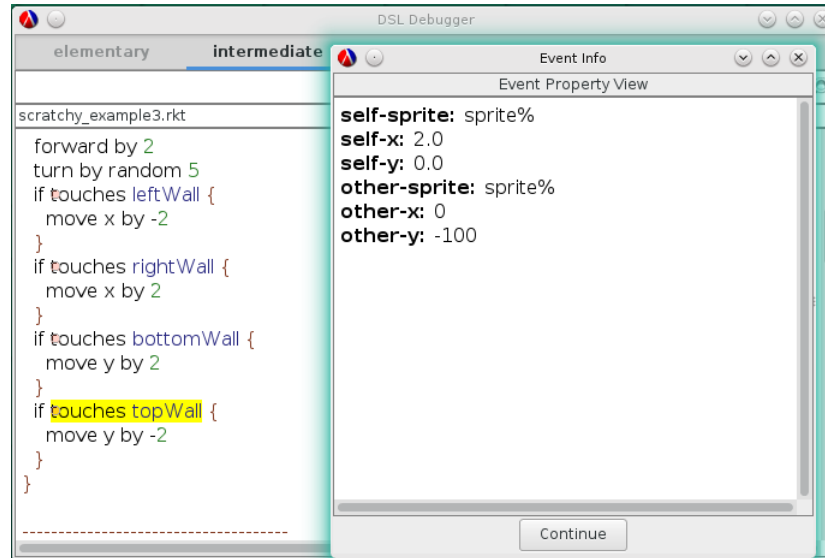


Figure 6.11. Stepping to the top wall statement

is 100, the top wall's y value should be larger. Therefore, through the event information presented in the pop-up window, the programmer was able to detect the unexpected behavior and find the problem faster with just four steps. The intermediate mode enabled to choose an event that would be relevant for a debugging task and presented a more useful, less noisy debugging interface.

6.2 POP-PL Debugger

POP-PL (Florence et al. 2015) is a message-based, declarative DSL to help a doctor describe and automate medical treatment. The POP-PL debugger contains two debugging modes, an elementary mode and an advanced mode. In the *elementary mode* (Figure 6.12), there are a source view (left), a message view (top-right), and a handler view (bottom-right). Messages sent and received during the execution of the POP-PL program are appended to the message view. The *message view* uses a hierarchical, collapsible tree structure to visualize messages, where a parent-children relationship denotes message-send causality. The parent node represents a message received, and the child node represents an outgoing message resulting from the incoming message. In the figure's message view, the [checkaptt] message was clicked (marked in blue), and the handler view was immediately updated to display the name of the handler that was responsible for the clicked message and to display the message histories associated with a handler. The *handler view* displays the handler name

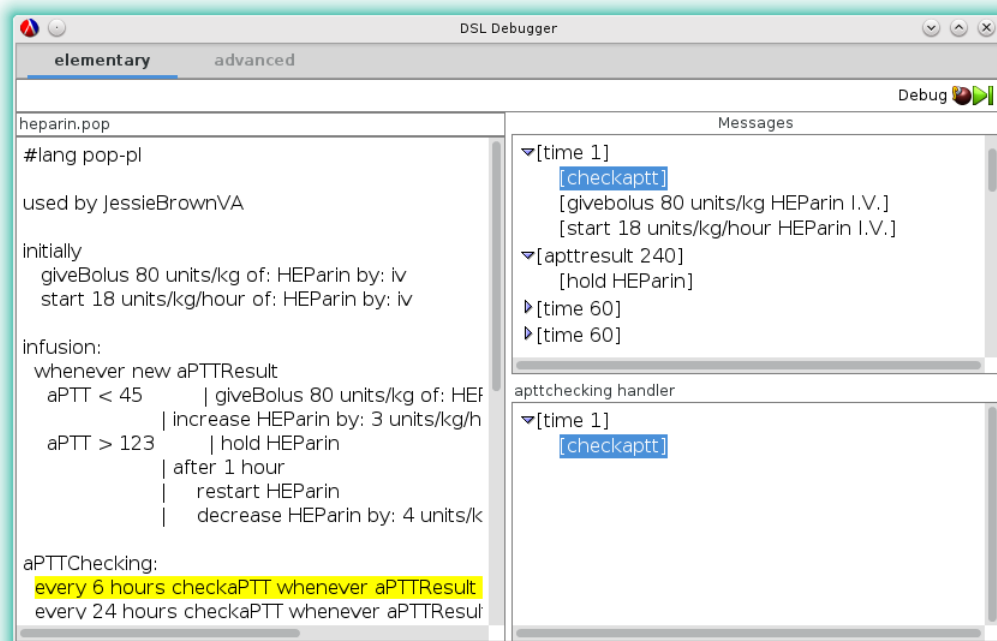


Figure 6.12. POP-PL debugger interface in an elementary mode

in the title bar of the handler view (in this example, `apttchecking`) and also uses a hierarchical tree structure to visualize the message-send causality. Clicking the `[checkaptt]` message in the handler view navigates to the origin of messages in the source view. In the source view, the line

```
every 6 hours checkaPTT whenever aPTTResult outside of 59 to 101, x2
```

was highlighted in yellow, which indicated that the `checkaptt` message was caused by this instruction.

The *advanced mode* (Figure 6.13) provides a debugging language to assist a fully programmatic control and a timeline view to visualize the events in a temporal fashion (obtained by clicking the *Timeline* button in the top-right corner). Because the advanced mode targets a user who is proficient in programming (either a traditional programmer who is used to programming in GPLs or a domain expert who is comfortable with programming), the debugging language is an extension of a GPL with primitives available in the intermediate mode. In the script view, the debugging language can manipulate events provided by the debugger implementation.

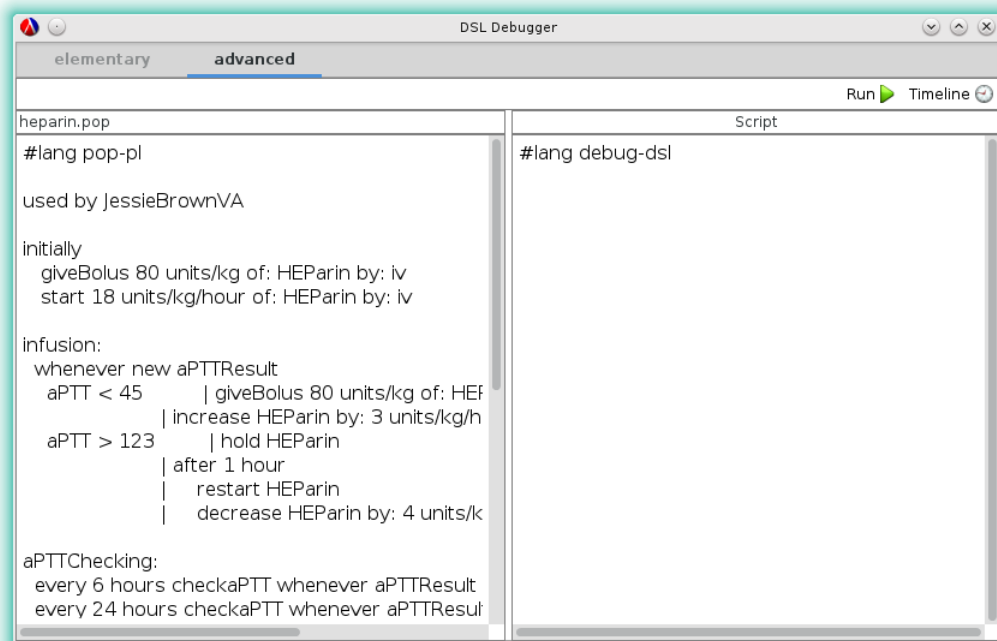



Figure 6.13. POP-PL debugger interface in an advanced mode

For the POP-PL debugger, `receive-msg-e` and `send-msg-e` are events available in the advanced mode. The `receive-msg-e` event represents an incidence of an incoming message, and the `send-msg-e` event represents an incidence of an outgoing message.

6.2.1 Finding POP-PL Bugs in the Elementary Mode

The execution of the “`heparin.pop`” program shown in Chapter 4 produced a failure message:

```
-----
/home/xiangqi/work/6racket/racket/racket/share/pkgs/pop-pl/examples/heparin/heparin.pop >
/home/xiangqi/work/6racket/racket/racket/share/pkgs/pop-pl/examples/heparin/heparin.pop
/home/xiangqi/work/6racket/racket/racket/share/pkgs/pop-pl/examples/heparin/heparin.pop
FAILURE
name:      check-match
location:   heparin.pop:30:14
params:     #f
actual:      '([restart HEParin] [decrease HEParin 4 units/kg/hour])
expected:    '(list-no-order
  (message (list-no-order 'restart _ ____ ) (list (eq heparin)) _))
  (message
    (list-no-order 'decrease _ ____ )
    (list (eq heparin) (eq (-number 3 units/kg/hour)))
    _))
 Check failure
-----
0 success(es) 1 failure(s) 0 error(s) 1 test(s) run
```

Because the “`heparin.pop`” program contains the following unit tests at the end of the program:

```
--- Tests ---
[giveBolus 80 units/kg of: HEParin by: iv]
[start 18 units/kg/hour of: HEParin by: iv]
[checkaPTT]

> aPTTResult 240
[hold HEParin]

> wait 1.5 hours
[restart HEParin]
[decrease HEParin by: 3 units/kg/hour]
```

The unit tests specify that a message of `[decrease HEParin by: 3 units/kg/hour]` in response to the `wait 1.5 hours` message is expected, but the failure message showed “`actual: '([restart HEParin] [decrease HEParin by: 4 units/kg/hour])'`”, which means that the actual message was `[decrease HEParin by: 4 units/kg/hour]`.

To debug the program, the programmer used the elementary mode to find and click `[decrease HEParin by: 4 units/kg/hour]` to understand why this message was sent. The handler view was updated showing that the message was sent by a `g23` handler, which means that the handler was an anonymous handler because the name was nonexistent in the program. As shown by Figure 6.14, the highlighted message in the handler view was clicked to find the message origin, and decrease

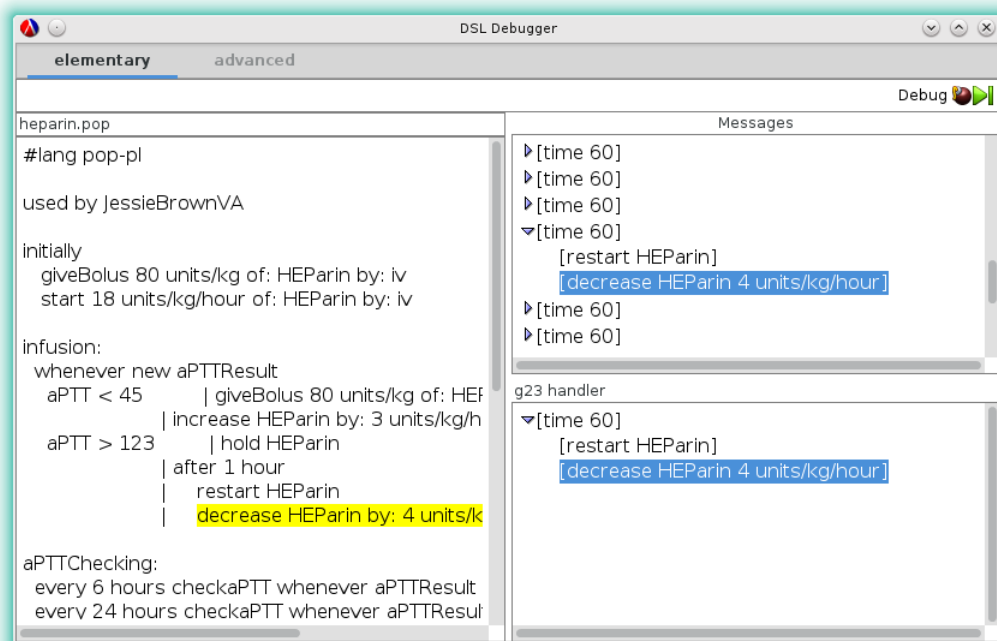


Figure 6.14. Debugging scenario in the elementary mode

HEParin by: 4 units/kg/hour inside a whenever block was highlighted. By looking at the code, the programmer realized that he made a typo. The highlighted code should be [decrease HEParin by: 3 units/kg/hour] instead.

6.2.2 Finding POP-PL Bugs in the Advanced Mode

The execution of the “popa.pop” program produced a failure message:

```
-----
/home/xiangqi/work/6racket/racket/racket/share/pkgs/pop-pl/examples/popa/popa.pop >
/home/xiangqi/work/6racket/racket/racket/share/pkgs/pop-pl/examples/popa/popa.pop
/home/xiangqi/work/6racket/racket/racket/share/pkgs/pop-pl/examples/popa/popa.pop
FAILURE
name:      check-match
location:  popa.pop:34:14
params:    #f
actual:    '([notifydoctor]
[notifydoctor]
[notifydoctor]
[notifydoctor]
[notifydoctor]
[notifydoctor]
[notifydoctor]
[notifydoctor]
[notifydoctor])
```

The failure message indicated that the unit tests at the end of the program were not satisfied due to the actual list of [notifydoctor] messages. The “popa.pop” program is:

used by OSFSaintFrancis

initially

start 25 micrograms/hour of: fentanyl
set onDemandFentanyl to: 25 micrograms

pain:

notifyDoctor whenever painscore > 5, x2, 1 hour apart,
since last notifyDoctor

minorPain:

whenever painscore < 5, x2, 30 minutes apart
notifyDoctor

--- Tests ---

```
[start 25 micrograms/hour of: fentanyl]
[set onDemandFentanyl 25 micrograms]
> painscore 9
> wait 61 minutes
> painscore 4
> wait 31 minutes
> painscore 6
> wait 61 minutes
```



```

> painscore 7
> painscore 2
> wait 31 minutes
> painscore 3
[notifyDoctor]
> wait 61 minutes

```

In the unit tests, the messages inside the brackets denote messages that should be sent from the medical system, and the messages after the > prompt denote messages being sent to the system.

The elementary mode was used to debug the program first, but the message view was filled with a lot of messages because of many internal [time 60] messages sent every 60 seconds by the system.

To make sure that the program was behaving correctly, the programmer started with a verification that the `notifyDoctor` message is only sent when the `painscore` message is sent twice and messages have values both above a threshold of 8. Because extracting meaningful messages and analyzing the relationship between messages were difficult, the programmer used the advanced mode to write a script to describe messages of interest:

```

(define-event pain-e receive-msg-e
  #:when (equal? (attr receive-msg-e 'type)
                 'painscore)
  #:when (> (car (attr receive-msg-e 'values)) 8))

(define-event notify-e send-msg-e
  #:when (equal? (attr send-msg-e 'type)
                 'notifydoctor))

(timeline pain-e)
(timeline notify-e)

```

In the above example, the `pain-e` event is defined in terms of the provided `receive-msg-e` event specifying that the type of the `receive-msg-e` event should be `'painscore` and that the value of the event should be over 8. The `notify-e` event describes the occurrence of the `notifyDoctor` message, which is defined in terms of the provided `send-msg-e` event. The two `timeline` statements assists visualizing the pattern of events in terms of timelines in the event view. By clicking the *Run* button, the debugging script was run, and the event view was opened by clicking the *Timeline* button. Figure 6.15 shows the resulting interface. The event view gives an overview of events first and allows exploration of events by mouse-hovering over the circles. Each

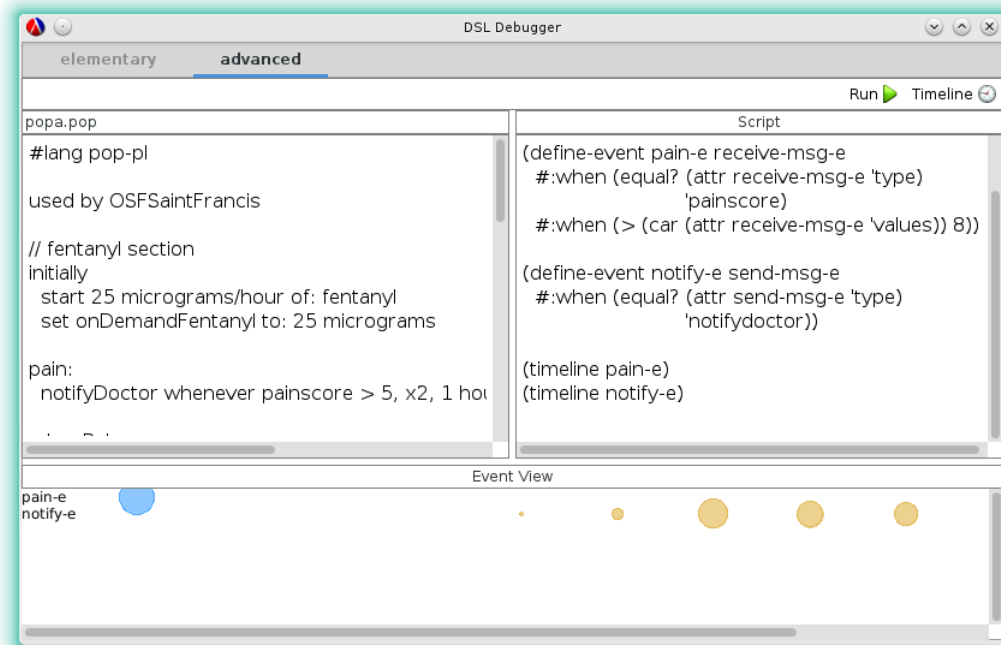


Figure 6.15. Debugging scenario in the advanced mode

circle represents an event or a group of events, and time increases moving left to right. The radius of a circle at a particular interval in time is proportional to the interval's percentage of events for the entire range of time.

The programmer checked the detail about the `pain-e` event circle and found that only one event instance was triggered. The next event circle happening after the `pain-e` event was a `notify-e` type, and the detail is shown in Figure 6.16. Clicking the event entry in the *Event Info* window navigated to the source context of the event (highlighted in yellow), which indicated that the event was sent by a `pain` handler. Because this program was expecting that a `notify-e` event was triggered only after two instances of `pain-e` events, the timeline view showed an erroneous behavior. By looking at the highlighted code in the source view, the programmer realized that the `notifyDoctor` message would be sent if the `painscore > 5 ,x2 ,1 hour` apart condition is true, but the desired condition for the `painscore` should be `painscore > 8` as expressed by the debugging script.

Therefore, the programmer found the mistake, and the `pain` handler should be written as:

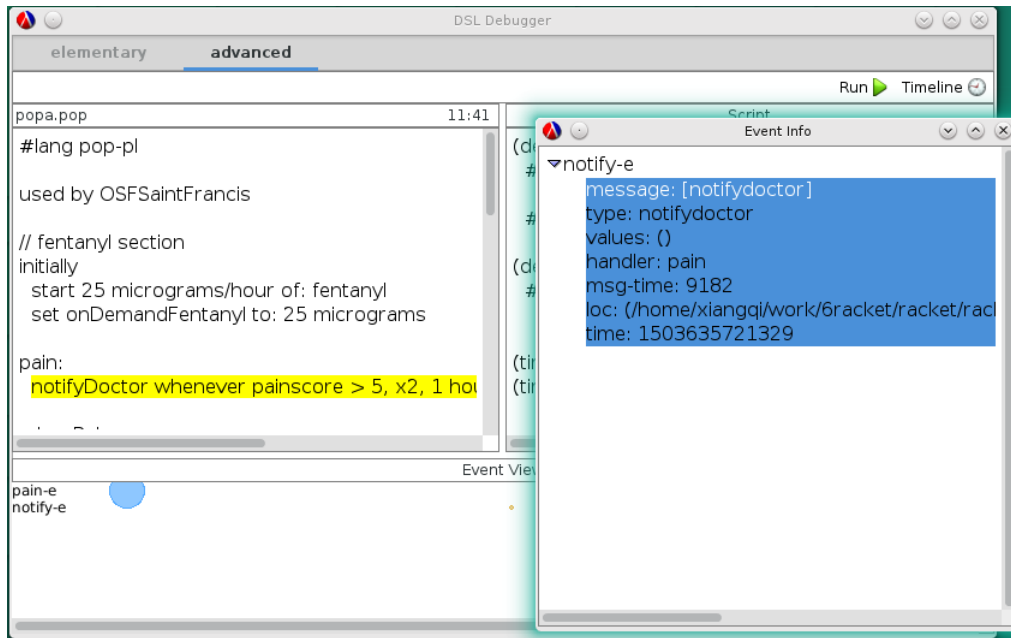


Figure 6.16. Exploring with timelines in the advanced mode

```

pain:
  notifyDoctor whenever painscore > 8, x2, 1 hour apart,
  since last notifyDoctor

```

6.3 Medic Debugger

As described in Chapter 3, Medic is a metaprogramming language describing program transformation without modifying the original source code. The Medic debugger interface is shown in Figure 6.17. The *step view* contains the same buttons as the step view in the Scratchy debugger, but the stepping operation works with each *match-form* in Medic (the current *match-form* under evaluation is highlighted in yellow). At each step, the *event property view* displays debugging information related to the execution effect of *match-form*. In the event property view, the *module-name* represents the module path of the source program to be transformed by the Medic program, and the *target-locs* represents a list of positions (a pair of line number and a column number) to be inserted with the *to-insert* code. The *port* and *scope* indicate the insertion scope. In the right source view, the insertion points are marked by yellow triangles in a straightforward way.

6.3.1 Demonstration of Finding Bugs

In Figure 6.17, the `target-locs` in the event property view showed that two locations would be affected by the first *match-form*, which was not an intended behavior. The programmer just wanted to add a `(printf "table =~v\n" dependency-table)` statement to the location inside an `append-export-entries` function in the “src.rkt” program, but the right-side source view showed that the first *match-form* would also insert the `printf` code into a matched location inside a `set-dependency-table` function. The `append-export-entries` function is:

```
(define (append-export-entries current-layer)
  (let-values ([lower-layer lower-exports]
               (get-lower-exports current-layer)))
  (for ([e (in-list (hash-ref export-table current-layer '()))])
    (let ([e-pair (cons e current-layer)])
      (unless (hash-has-key? dependency-table e-pair)
        (let ([res (member e lower-exports)])
          (when res
            (let ([new-v (if (zero? current-layer) e (cons e lower-layer))])
              (hash-set! dependency-table e-pair new-v))))))))))
```

Before making any changes to the first *match-form*, the programmer went on to check the correctness of other *match-forms* by clicking the *Step* button to examine the insertion effects. After stepping to the last *match-form*, the programmer was sure that the program only contains a bug with the first *match-form*. Because the first *match-form* would match two locations in the “src.rkt” program, the scope of the `at` form was constrained by adding a new `#:before` option:

```
[at (hash-set! dependency-table e-pair new-v)
  #:before (append-export-entries current-layer)
  [on-exit (printf "table =~v\n" dependency-table)]]
```

which specifies that the `printf` code will not be inserted until there is a code segment matching `(append-export-entries current-layer)` before the insertion point at the exit of the statement, `(hash-set! dependency-table e-pair new-v)`. By debugging the modified program, the programmer confirmed that the new change fixed the bug.

CHAPTER 7

EVALUATION

This chapter evaluates whether Ripple enables more useful and easier debugger construction. Specifically, I evaluate the support for building domain-specific debuggers, the support for building effective debuggers, and the support for reducing the debugger construction effort. Since Ripple is not yet in wide use, this chapter cannot report practical experiences. However, to attempt a robust evaluation, I tested on the three DSLs presented in Chapter 6: Scratchy, POP-PL, and Medic.

7.1 Support for Domain Customizations

Since DSLs focus on domain problems, an ideal DSL debugger should have an interface mapped to the domain concepts and notations as closely as possible. To support domain customizations, Ripple provides means of customizations in the back end as well as the front end where the back end relies on domain-specific events and the front end relies on flexible views.

Domain customizations for the three example DSLs are shown in Figure 7.1.

Language	Domain Concepts	Domain Customizations
Scratchy	sprite, sprite manipulation	elementary mode: step view, variable view (Figure 6.2); intermediate mode: touch-e (Figure 6.3)
POP-PL	prescription, actors, messages, handlers	elementary mode: message view, handler view (Figure 6.12); advanced mode: send-msg-e, receive-msg-e (Figure 6.15)
Medic	layer, debugging code insertion, location specification	elementary mode: step view, event property view, debug program view (Figure 6.17)

Figure 7.1. Ripple support for domain customizations

By taking into account the possible debugging needs of a DSL and the programming background of DSL users, each DSL is enriched with different debugging modes. Like any debugging tool, familiarity with debugger features is a prerequisite to debugging. Without regard to the features of the debugger, I assume a debugger to be *domain-specific* if the knowledge required to use and understand the debugger falls within the domain knowledge of a DSL user. For the elementary mode, beyond the debugging knowledge, the knowledge required for each DSL debugger involves:

- Scratchy: the variable view requires a user to understand the interpretation of sprite-state variables, which are closely related to the domain concepts.
- POP-PL: a user needs to know that the message view visualizes the message history in the clinical network and the handler view visualizes the message history of a handler, where the concepts of messages and handlers are within the domain.
- Medic: a user can easily understand the interface where the source view and the debug program view annotate the entities on which the user is working with visual marks to help program comprehension. For the event property view, a user needs to understand the meaning of each property, but these properties can be easily mapped to domain concepts.

Overall, one can see that domain-specific debuggers are achievable for DSLs with Ripple. As for the intermediate and advanced mode, due to a concern with a more flexible, expressive control, a significant amount of debugging knowledge is involved, and the only opening for customizations is events. To bridge the gap between debugging events and domain concepts, debugger developers can define domain-specific events and enable the use of domain-specific events for the two modes. However, a user needs to understand the mapping between domain-specific events and domain concepts and the interpretation of event-specific attributes.

7.2 Support for Effective Debuggers

The evaluation of the support for effective debuggers consists of the discussion of debugging support available via Ripple and the demonstration of Ripple's abilities to construct debuggers that are able to find bugs.

Although featuring domain-specific abstraction and notations, a DSL, by nature, is still a recognizable programming language. I make the assumption that GPL debugging techniques can be adaptable to DSLs on the condition that they do not sacrifice domain-specific concepts. As different

debugging techniques have their advantages and disadvantages in different application domains, finding a universally powerful collection of debugging techniques is difficult.

However, Zeller (2005) points out that debuggers generally provide the following functionality:

- *Execute the program and make it stop on specified conditions*
- *Observe the state of the stopped program*
- *Change the state of the stopped program*

I designed debugging features in Ripple according to the above guidelines. Due to the limitations and complications surrounding the functionality to “change the state of the stopped program,” Ripple replaces it with a navigation facility. In the realm of GPL debugging, large temporal or spatial chasms between the root cause and the symptom of an erroneous program account for a significant amount of debugging difficulty (Eisenstadt 1997), so Ripple puts special effort into providing tool support for navigation. In Ripple, each event instance carries an attribute that records the source location of an event, and graphical tools support event navigation to its source context.

The debugging facilities Ripple provides range from graphical control to programmatic control. Execution control can be achieved through event-by-event stepping, state observation is enabled through a user interface with customizable graphical views, and navigation comes with the connection between graphical entities and source context. In addition, the programmatic control allows open-ended debugging features (e.g., creating problem-specific operations such as tracing a variable or asserting a program invariant). However, all debugging support provided by Ripple is based on the events designed and developed by debugger implementers. Debugger implementers should have a good knowledge about a DSL design and possibly be informed about common errors associated with a DSL.

In evaluating the effectiveness of debuggers constructed through Ripple, there are two problems. First, effectiveness depends on the provided debugger features. For example, if debugger developers implemented a stepping-based debugger for a reactive programming language such as POP-PL, the debugger is unlikely to be effective. Second, effectiveness depends on the user groups. Users have varying programming skills and varying preferences towards different debugger features. For end users, graphical control will be considered more useful than programmatic control.

As a quantitative analysis of the effectiveness of debuggers requires considerable design of experimentation, I conducted a qualitative analysis of a debugger’s abilities to find bugs. Specif-

ically, since Ripple supports building a domain-specific debugger with more than one debugging mode, I evaluate the merits of the debugger being domain-specific and the debugger having different debugging modes according to my own debugging experiences.

To evaluate the benefits of domain-specific debuggers, I wrote a small Scratchy program with an error in the program. I tried to use the traditional, stepping-based GPL debugger in DrRacket, which would make a good candidate for a non-domain-specific debugger, to debug the Scratchy program, but the stepping facility did not work. Because the GPL debugger expands the DSL program into a program with only a small set of Racket kernel constructs, there is a one-to-many mapping between a DSL's constructs and kernel constructs, which causes conflicts with the original stepping mechanism. I decided to make modifications to the GPL debugger to print out the program states. However, the program output state consisted of low-level variable bindings existing at the level of the kernel language:

```
(let-values ([ (temp8) (%app find-method/who
                             'send temp9 temp7)])
  (let-values ([ (send-arg11) '4])
    (if (%app wrapped-object? temp9)
        (if temp8
            (%app temp8 (%app wrapped-object-neg-party temp9)
                        (%app wrapped-object-object temp9)
                        send-arg11)
            (let-values ([ (temp9) (%app wrapped-object-object
                                         temp9)])
              (%app (%app find-method/who 'send temp9 temp7)
                    temp9 send-arg11)))
        (%app temp8 temp9 send-arg11))))
```

The above output is generated by the Scratchy's `forward by 4` statement. The variable bindings were `temp9` with a value of a sprite object and `temp7` with a value of `'forward`. The low-level variables in a non-domain-specific debugger were not very useful for understanding program execution.

Debugging with the domain-specific debugger was easier. As seen in the demonstration in Section 6.1, I was able to see high-level program states, which enabled better program comprehension.

To evaluate the effectiveness of each debugging mode, I experimented with different debugging modes and demonstrated the experiences in Chapter 6. The overall experiences are:

- Scratchy: I found a bug in a program with the elementary mode. I also tried to debug another program with the elementary mode, which took 4 attempts to find the bug. However, with

the intermediate mode, I was able to select events with information in which I was interested, which generated more useful, less noisy debugging information.

- **POP-PL:** I demonstrated finding a bug by checking message histories in the message view and navigating to the source context from the message entry in the handler view in the elementary mode. I also tried to debug another POP-PL program with the elementary mode, but the messages shown in the debugger were overwhelming. I switched to the advanced mode and was able to write a small script to perform the desired operations.
- **Medic:** Because the Medic program was relatively small, the elementary mode was very straightforward for showing the effect of program execution.

In conclusion, from the demonstration of finding bugs, there is little doubt that the domain-specific debuggers built on Ripple are useful for DSLs, but there are no absolute advantages and disadvantages for each debugging mode. Depending on the debugging scenario, one debugging mode might be more effective than another. Generally speaking, graphical control in the elementary mode is effective for debugging programs needing graphical visualizations and interactions, and the programmatic control is useful for complicated, variable bug queries and explorations. Different modes can also be used together to achieve a more efficient experience.

7.3 Ease of Debugger Construction

Ripple places great emphasis on reducing the effort of debugger construction, and the emphasis drives the design principles of the debugging framework.

- **Reuse of debugging techniques.** Much like each DSL has a strength in solving problems in a particular domain, there exists a suitable debugging technique for each language. By encouraging and providing a library of DSL-friendly debugging techniques in the framework, developers get guidance in choosing and reusing appropriate techniques. So far, my system is still under development and just provides event-based support, but more useful debugging techniques can be added such as omniscient debugging since events allow a flexible extension of features. The debugging language provided by the system enables reuse of debugging techniques where debugger developers do not need to design and develop a scripting language to support programmable activities.

- **Reuse of debugging implementation.** Ripple supports associating a language with a debugger where a specification of debugging support (through events) is integrated with macro transformations. When a language is implemented in terms of another language, the language can reuse the debugging support available in the other language by reusing the debugging events. Specifically, the language can define events by filtering, combining, and transforming other languages' events. Instead of working with common, low-level events such as core events, a high-level language can just be concerned about its immediate, low-level events, which would contain debugging information sharing closer abstractions.
- **Reduction of user-interface development effort.** To reduce the effort of implementing a user interface, Ripple separates tool support into three debugging modes. Developers can just focus on one mode and learn to use related tools to implement a debugger. The elementary mode allows a composable interface to further divide debugger construction work into views where developers can reuse system views or implement customized views incrementally. Since Ripple provides automatic support for the intermediate and the advanced modes, if a DSL does not need graphical control for debugging, the interface development is minimal, and a debugger can be easily developed by defining debugging events in the back end.

According to my experiences with constructing Scratchy, POP-PL, and Medic debuggers, the effort involved in debugger construction consists of several nontrivial tasks, learning the DSL's domain concepts, designing a debugger's features, learning the language's macro implementation, and writing the debugger implementation. As the first three components are beyond my debugging framework's concern, I only measure the effort involved in debugger implementation. Though efforts can be measured by a time cost, I could not accurately keep track of the hours spent in debugger construction because of iterative, selective implementation of debuggers. For example, sometimes I just suspended the whole debugger implementation to learn more about a DSL design. Sometimes the debugger features were not satisfactory after use, and I restarted the process of a debugger implementation.

Figure 7.2 shows the size of debugger implementation in lines of code. The size of event instrumentation is relatively small. The domain-specific events I created for each DSL are: `construct-e` with customized bindings and `touch-e` for Scratchy, `send-msg-e` and `receive-msg-e` for POP-PL, and `module-entry-e` and `insert-e` for Medic. In Scratchy, the `construct-e` events are

Language	Lines of Code	
	Event	Interface
Scratchy	18	96
POP-PL	12	109
Medic	11	16

Figure 7.2. Size of debugger implementation

implemented by defining embedded events and involve the `construct-e` host events as well. For the `touch-e` event in Scratchy and other DSLs' events, events are defined as explicit events in terms of the lower-level language's `construct-e` events. The interface implementation (only the elementary mode is needed) remains the majority of work. But the total implementation size for each DSL is still relatively small compared to implementing debuggers from scratch. Because all views of the Medic interface reuse the system views, the interface implementation size of Medic is smallest, which indicates that a debugging framework should provide more tool support for interface implementation.

From the experiences of debugger construction, the event language in the back end is helpful for event instrumentation, and the tool support in the front end can reduce the cost of interface implementation. Overall, Ripple is useful in easing the development of a domain-specific debugger.

7.4 Performance

I examined the performance of debuggers according to time and space cost metrics. Time overhead added by debuggers includes the back-end event registration, the core debugger instrumentation and core program execution, and event mapping. I experimented on three DSL programs and measured the debugger cost compared to the original time cost. In Figure 7.3, the program size is measured by lines of code, and one can see that the time cost varies with DSLs. To investigate the cause of the variance of time, I checked the percentage of language constructs in a DSL program that would be affected by event instrumentation. As shown by the right-hand-side chart, the debugger time cost, which is measured by a percentage of increase in program execution time, is roughly positively correlated with the percentage of program code that is affected by event emission at run time.

Figure 7.4 shows the memory usage with a debugger and without a debugger, which measured

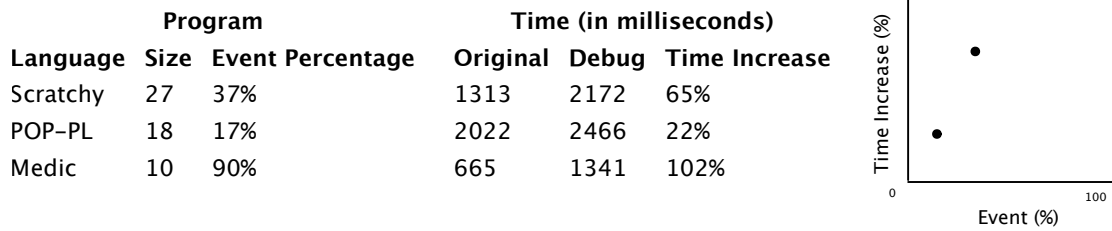


Figure 7.3. Time overhead of debuggers

Language	Megabytes	
	before	After
Scratchy	243	384
POP-PL	200	422
Medic	188	383

Figure 7.4. Memory usage

in terms of total heap allocation. Since debugging information is encapsulated in event attributes, depending on how a DSL debugger interface implementation handles event information, the space cost varies. For example, because the debugging session of POP-PL stores all the message histories of a handler, the POP-PL debugger uses the most memory compared to the Scratchy and Medic debuggers. However, Ripple also tries to optimize the space cost by reducing some avoidable memory cost such as only generating needed events for a DSL debugger.

In summary, the time and space costs associated with debuggers are largely affected by events, and the performance of these debuggers is reasonable.

CHAPTER 8

OUTLOOK

I have presented an approach to debugging DSLs defined with macros. Instead of relying on the debugging support provided for the host language, I employed an event-oriented approach that relies on events to capture run-time program states and to enable debugger inspection and manipulation. I provided a core programming-language model and event constructs for mapping the evaluation of a DSL to domain-specific events and further designed and developed a debugging framework on top of the domain-specific events. I implemented the debugging framework, Ripple, and used Ripple to construct various domain-specific debuggers. From my debugging experiences, I demonstrated that Ripple is useful in easing the development of effective, domain-specific debuggers.

My debugging framework focuses on a macro-expansion view of language implementation and provides event support that can be integrated with macro transformations. However, for DSLs that are implemented through means other than macros, the general event design is applicable if the DSLs can be instrumented with debugging events. The core model and events are vital in event mapping, and my core programming-language model can serve as a good target for compilation of a broad range of programming languages. For DSLs that are implemented on other platforms and are compiled to other host languages, the host languages need to be able to be mapped to the core model. The constructs specifying the event mapping across language layers are mostly platform independent and can be adapted easily to a different platform. The categories of event mappings are universal to all systems. In addition, because the front-end design for the debugging framework just works with events, the architecture of the debugging framework and the front-end design is adaptable to other systems.

8.1 Composition of DSLs

A DSL is usually designed to solve a problem in a particular domain, and sometimes a complicated problem spans many domains requiring coordination of multiple DSLs. Composition of DSLs includes syntactic and semantic composition where the syntactic composition allows a

combination of constructs from different languages and where the semantic composition deals with the behavioral semantics. Since Ripple solves the problems of debugging macro-based DSLs, I will focus on the syntactic composition of multiple DSLs enabled by macros. Because macros are composable, multiple DSLs can coexist in a larger application along with the host language. This section discusses the issues of debugging coexisting DSLs.

8.1.1 Debugging Coexisting DSLs

Suppose that we have three DSLs dealing with different problem domains in the field of digital electronics:

- The Gate language handles the construction of basic logic gates including AND gates, OR gates, and NOT gates. The Gate language provides `define-gate`, `set-gate-input`, and `get-gate-output` constructs.
- The Device language handles the construction of digital devices that are abstractions of wired logic gates. A digital device can be either an adder or a comparator, and each digital device allows 32 bits input and output. The Device language provides `define-device`, `set-device-input`, `get-device-output`, and `overflow?` constructs.
- The Circuit language handles the connection of digital circuit entities and provides a `wire` construct.

If we want to combine logic gates and digital devices to construct an electrical circuit, we can write an application that uses constructs from the Gate, Device, and Circuit languages. The following program is an example of DSL composition:

```
(define-gate and-gt [type 'and])
(define-device adder [type 'adder])
(define-gate not-gt [type 'not])

(wire [from (device adder [bit-at 32])]
      [to (gate and-gt 'input1)])
(wire [from (gate and-gt)]
      [to (gate not-gt 'input1)])

(set-device-input adder [input1 "11100011"] [input2 "00111011"])
(set-gate-input and-gt [input2 1])
(get-gate-output not-gt)
```

In this example, an AND gate, an adder device, and a NOT gate are constructed. The first wire statement wires `adder` and `and-gt` together and connects the 32nd bit of `adder`'s output to the first input port of `and-gt`. The second wire statement wires `and-gt` and `not-gt` together by connecting the output of the `and-gt` gate to the first input port of `not-gt`. The `set-device-input` initializes the two input ports of the `adder` device, and the `set-gate-input` adds an input value to the second input port of `and-gt`. The `get-gate-output` statement obtains the output value of the `not-gt` gate.

Suppose that there exist three domain-specific debuggers for the three DSLs, which are built using the Ripple debugging framework. The following domain-specific events are created for the debugging support:

- `gate-e` to denote the occasion of a new logic gate construction, `update-gate-e` to denote the occasion of setting a gate's input, and `output-e` for accessing a gate's output.
- `device-e` to denote the occasion of a new digital device construction, `update-device-e` to denote the occasion of setting a device's input, and `output-e` for accessing a device's output.
- `connect-e` to denote a wire connection made between two digital circuit entities.

The three domain-specific debuggers are:

- The Gate debugger (Figure 8.1) consists of a step view that provides execution control and a stepping facility, a gate view that displays the gate information including gate names, gate types, and gate input, and an output view that displays the output of gates. In the gate view, any newest input changes are marked by asterisks.
- The Device debugger (Figure 8.2) consists of a step view that provides execution control and a stepping facility, a device view that displays the device information including device names, device types, and device input, and an output view that displays the 32-bit output of devices. In the device view, any newest input changes are marked by asterisks.
- The Circuit debugger (Figure 8.3) consists of a step view that provides execution control and a stepping facility and a connection view that displays the information about the two connecting entities.

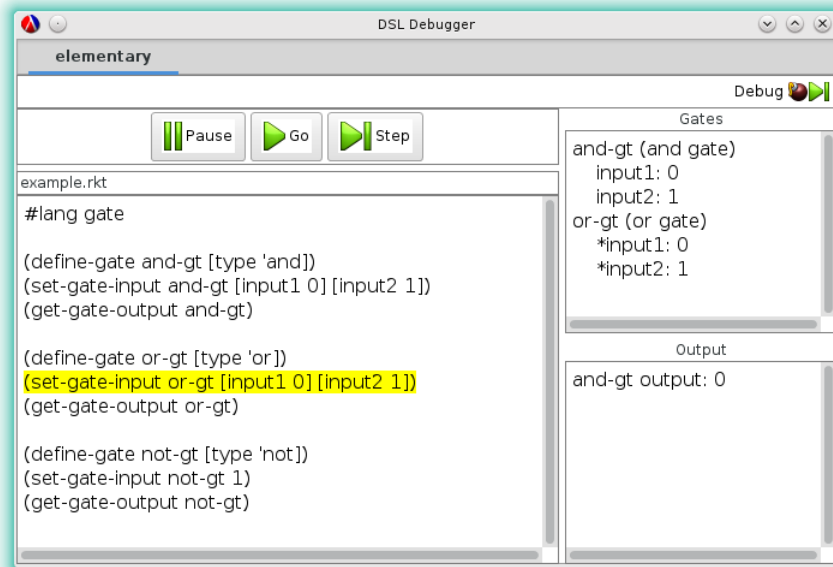


Figure 8.1. Gate debugger

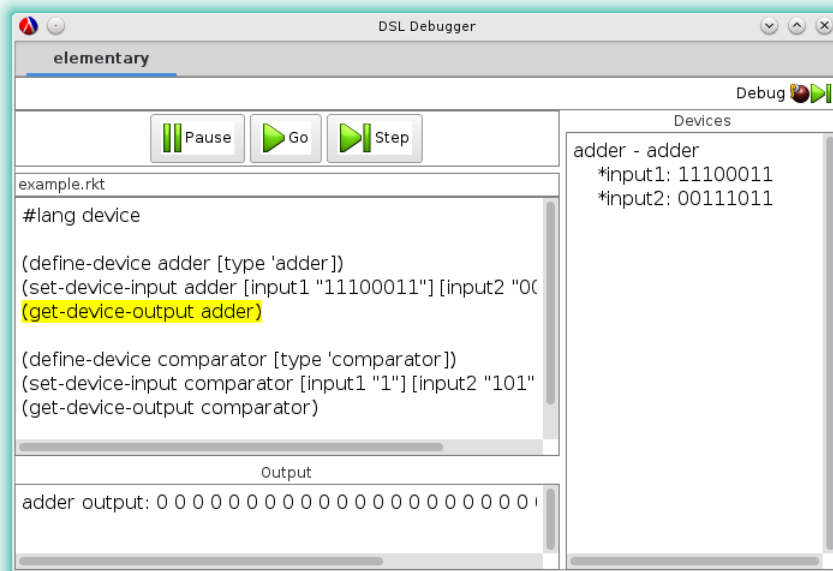


Figure 8.2. Device debugger

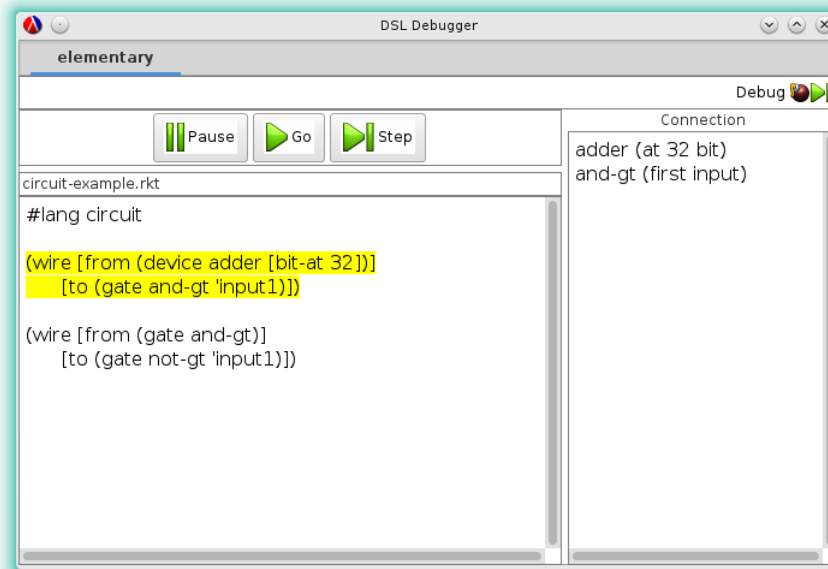


Figure 8.3. Circuit debugger

To debug coexisting DSLs, there are two approaches. First, treat the composing languages as a unit and develop a specific debugger for the whole unit. Second, treat the composing languages as a combination of sub-languages and reuse the debugging support developed for each sub-language. Since DSLs can be combined in different ways, which adds difficulty to implementing debugging support, I choose the latter approach for debugging support for coexisting DSLs.

The event design in Chapter 4 and the architecture of the whole debugging framework in Chapter 5 were originally designed for debugging a single DSL in a debugging session. However, the design for the debugging events and debugging framework works for debugging multiple DSLs at the same time as well. Each DSL can still preserve its event instrumentation and front-end interface implementation to accommodate composition of DSLs. For example, the Gate language can still keep its `gate-e`, `update-gate-e`, and `output-e` events and its debugger interface. Even though the Gate language and the Device language both provide an `output-e` event, the event name conflicts pose no problem for the composition of DSLs. Each event instance in my event design contains a *layer* field (see *evnt* in Figure 4.8), which associates an event with its enclosing module path and can distinguish events generated from different DSLs.

Ripple is an implementation of the debugging framework presented in Chapter 5, which involves event generation and interface presentation and has debugging a single DSL in mind. When a user wants to debug a DSL program, the DSL debugger is shown with the system views first initialized. The layout of the custom views written by debugger implementers is not initialized until the click of the *Debug* button in the DSL debugger to start a debugging session. To enable multiple DSLs in a debugging session, the debugger interface needs to be switched according to the run-time evaluation context so that the correct debugger interface is activated. Therefore, the interface presentation in Ripple needs to be changed to coordinate the dynamic switch of debugger interfaces, but the algorithms about event generation can remain unchanged.

At run time, the evaluation of a certain construct from a DSL can generate a debugging event, and a debugger interface for a DSL is changed if the debugging event comes from a different DSL. To activate the right debugger interface, Ripple needs to find the interface layout specification for the current DSL, update the debugger interface with correct views, and activate the event handlers that belong to the DSL. For the previous example of composition of the Gate, Device, and Circuit languages, the debugger is shown in Figure 8.4 where the interface is updated to the interface belonging to the Device language when the debugger executes to the `set-device-input` statement.

8.2 Improving the Debugging Experience

Future work is needed to further improve the debugging experience. My debugging framework aims to reduce the cost of debugger construction by incorporating a suite of reusable tools, and more tool support can be added into the framework. For example, more sophisticated data visualizations are needed, and a means to support and ease visualization construction for users is desirable. Currently the debugger visualizations are either implemented by debugger developers or enabled by the debugging language, which does little to ease users' burdens in constructing visualizations. The design of the debugging language in the advanced mode is an extension of the GPL provided by a DSL-construction platform, which may add learning overhead if users switch to another debugging framework implemented on a different platform. Further improvements in the debugging language are necessary, for example, a design for a language-independent debugging language.

My debugging framework also needs to address the issue of language composition and provide better support. The approach of switching the debugger interface according to the run-time events seems straightforward, but it has limitations.

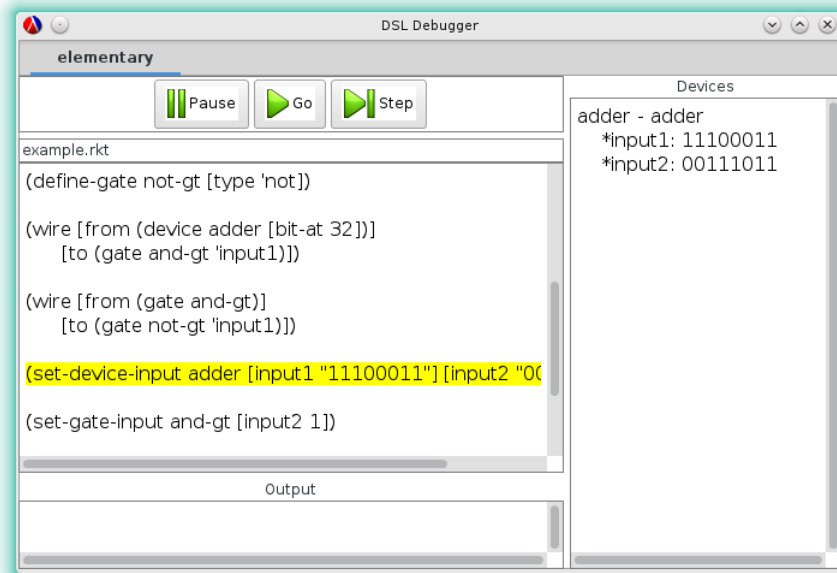


Figure 8.4. Debugger interface for composition of DSLs

First, the interface switch at run time may be confusing and distracting. For example, a user might be occupied with exploring a debugging interface belonging to a DSL and suddenly be presented with a different interface due to the activation of a new event.

Second, learning overhead increases with the number of DSLs. Different DSLs might have different interface layout and interface operations, and users need to be familiar with the features associated with each DSL to be able to understand and use the current active debugger.

Third, the approach works best with a sequential model of events. The approach assumes that only one event can be triggered at a time and therefore updates or switches the debugger interface according to the current event. If events can happen in parallel, a different approach will be needed.

The effectiveness of switching debuggers dynamically remains unclear. When the debugger is evaluating a construct belonging to a DSL, the debugger switches to a local debugger that is associated with the DSL and only displays the debugging information relevant to the DSL. In consequence, the connection between DSLs is unaddressed because the debugger is composed of independent, local debuggers obscuring the possible evaluation effects of one language on another within the whole application. However, there are a few means for possibly improving the debugging

experience.

First, for languages that have conflicting event names, the events sharing the names can either have the same semantics or different semantics. For example, the core language provides a `construct-e` event that represents the occasion of a single-step reduction of an expression, and Scratchy also provides a `construct-e` event having the same semantics. A kind of view for events having the same semantics is enough to display the debugging information, and the view can be placed in a fixed area in the debugger to reduce the cognitive load of context switching. For languages composed together, a fixed interface might be better for events from different languages sharing names and semantics. For example, we can have a fixed variable view to display the variable bindings carried by the `construct-e` events no matter which language the `construct-e` belongs to.

Second, the Ripple framework supports a debugging language for DSLs, which is provided by the advanced mode. A debugging language only works with events, which can eliminate the context switching problem associated with graphical interfaces. To support a debugging language for coexisting DSLs, event name conflicts have to be resolved to effectively refer to events without ambiguity. Events can be renamed to distinguish between events that have the same names in the composed DSLs.

CHAPTER 9

RELATED WORK

There is a large body of work on debugging techniques and tools for GPLs, and recently there has been an increase in debugging support for DSLs. Since this dissertation focuses on an event-based approach to DSL debugger construction, I describe the related work in three parts: event-based debugging support for GPLs, language workbenches' debugging support, and other tools' support for debugging DSLs.

9.1 Event-Based Debugging for GPLs

Coca (Ducasse 1999) is a debugger for C including an expressive breakpoint mechanism and trace analysis facilities. Coca models program execution as a sequence of events and supports debugging through querying the traces. The events are related to language constructs, and breakpoints are associated with events instead of source lines. As Coca is only concerned about debugging C programs, its event model is simple and tied to C constructs such as `for` and `if`. In comparison, the core events in my work are not tied to a particular language, and the core events are able to fully reconstruct the state of an evaluation.

RAIDE (Johnson 1977) offers a language-independent debugging language to enable debugging, and the debugging language involves some metaprogramming concepts and a model of events to control the execution. The events in RAIDE are associated with entry and exit to statements or routines, variable access and update, and other events related to the system. In my framework, a debugging language is also provided, though not language-independent.

Dalek (Olsson et al. 1990) presents an event-based approach to debugging with a debugging language and a model of events. Similar to my event model, Dalek supports primitive events and high-level events. The primitive events are similar to breakpoints requiring explicit event raises from users. High-level events provide means of expressing complex program behavior and are defined in terms of a list of constituent primitive events. By comparison, my system only supports defining primitive events by debugger implementers, and debugger users cannot change the primitive events

associated with a language.

UFO (Auguston et al. 2003) models debugging as a computation over event traces where events are modeled by time intervals instead of time points of execution. Each event has a beginning, an end, and a duration, and consequently events are related by a partial ordering and by inclusion. UFO presents a fixed set of event types such as types associated with whole program execution, expression evaluation, function call, or loop iteration. In contrast, the event model I used is based on time points of execution.

MzTake (Marceau et al. 2006) is a dataflow language that supports event-oriented debugging. MzTake places an emphasis on the expressiveness of a debugging language to manipulate events such as providing a collection of event-processing primitives and a means of abstraction. However, unlike the sophisticated event model in my dissertation, the event concept in MzTake is limited, and the debugging language can only work with events related to trace points of variables.

To monitor the behavior of a program, a generalized path expression (Bruegge and Hibbard 1983) is proposed to detect event occurrences where a path expression contains repetition, sequencing, and exclusive selection operators and can use predicates to further delimit events with regard to request, activation, and termination of events. Compared to the previous event-based systems, EBBA (Bates 1995) is closer to my model of events. EBBA views debugging as a process of building models of expected program behavior where the model is based on events. EBBA supports sequential, choice, concurrency, and repetition event expression operators for modeling the behavior of a program, which inspired my choice for event composition operators for high-level events. The EBBA tool also emphasizes using events for debugging information units where an event class has an event class name and a list of event attributes.

9.2 Debugging Support on Language Workbenches

Spoofax (Kats and Visser 2010), MPS (JetBrains 2004), MontiCore (Krahn et al. 2008), and xText (Efftinge and Volter 2006) are language workbenches that support language creation and common IDE services for domain-specific languages. To the best of my knowledge, MontiCore and xText do not consider debugging support for DSLs, and the debugging support provided by Spoofax and MPS remains ongoing work since finding a general solution for various DSLs is difficult.

An exploration of debugging support for Spoofax is based on the work of Lindeman et al. (2011), which aims to reuse the debugging support provided for the host language and proposes mapping a

DSL to four universal events: `step`, `enter`, `exit`, and `var` events. The `step` event is related to an expression evaluation, the `enter` and `exit` events are related to the changes of stack frames, and the `var` event is related to variable declarations. These four debugging events are heavily tied to concepts found in stepping-based debugging, which results in DSL debuggers that mostly support setting breakpoints, stepping, and observing program states in variable views. DSL debuggers cannot have flexible debugging views and operations.

MPS also tries to solve the problem of the abstraction gap between a DSL and a hosting GPL and supports building DSL debuggers with domain concepts. However, the debugging support is inadequate. MPS focuses on languages that translate to Java and tries to reuse the Java debugger support for DSLs. Therefore, the interface of custom debuggers on MPS is similar to the Java debugger, and the debugging operations are related to stepping-based techniques.

In summary, I did not find a language workbench that is able to support domain-specific debuggers with domain-specific views and operations. By comparison, my approach of mapping the execution of a DSL to domain-specific events enables truly domain-specific debuggers.

9.3 Other Debugging Support for DSLs

Debugging support for DSLs is relatively new and rare compared to debugging support for GPLs. One direction relies on the DSL grammar or language specification to generate DSL debuggers such as DDF (Wu et al. 2008), LISA (Henriques et al. 2005), and TIDE (Van Den Brand et al. 2005). In DDF, the DSL grammar is augmented with additional code to enable abstraction mappings from the GPL level to the DSL level, and the code can also be weaved into the grammars using an aspect-oriented approach (Wu et al. 2005). The technology presented in LISA is based on DDF. TIDE also uses events to enable debugging support, and events are inserted into a language specification that is based on ASF+SDF (van Deursen et al. 1996). These tools reuse the debugging tool support at the GPL level, and the features of the DSL debuggers are tied to the GPL debugger features such as stepping and setting breakpoints. Instead of relying on the stepping-based GPL debugging technique, my framework enables flexible debugging views and operations.

Another direction focuses on event modeling without language grammars, which is more similar to our approach. The moldable debugger (Chis et al. 2014) supports building a debugger that has flexible views and debugging operations. In the moldable debugger, debugging events can be customized to capture domain concepts by defining high-level events. The debugging events

provided by the moldable debugger are modeled as predicates over the program's run-time states, and primitive events are fixed including attribute read, attribute write, method call, message send, and state check. In contrast, my event framework does not provide a fixed set of primitive events for DSL event customization; I allow each language to have its own set of events that suits its language features and to enable domain-specific events through an event-mapping mechanism.

In the realm of domain-specific modeling languages, events are also used to observe and control the behavior of a model. BCOoL (Deantoni 2016) relies on domain-specific events to coordinate the behavior of heterogeneous languages. A variant of omniscient debugging (Bousse et al. 2015), which was proposed for executable domain-specific modeling languages (xDSMLs), uses events to capture execution states but only captures values of mutable fields and transition from one state to another state. In comparison, my core events aim at capturing whole machine states including information about continuations, and my event model is more general, offering a means to capture a variety of information depending on debugging needs. For nonexecutable models, model simulators and model transformations define the semantics of models, which can also be enabled with debugging support. Simulators can be instrumented with debugging operations (Van Mierlo et al. 2017), and an omniscient debugging technique can also be provided for model transformations (Corley et al. 2017).

REFERENCES

- Mikhail Auguston, Clinton Jeffery, and Scott Underwood. A Monitoring Language for Run Time and Post-Mortem Behavior Analysis and Visualization. In *Proc. Fifth Intl. Wksp. on Automated Debugging*, 2003.
- Eli Barzilay. The Scribble Reader. In *Proc. Wksp. on Scheme and Functional Programming*, 2009.
- Peter C. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. *ACM Transactions on Computer Systems* 13(1), 1995.
- Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, and Benoit Baudry. Supporting Efficient and Advanced Omniscient Debugging for xDSMLs. In *Proc. Software Language Engineering*, 2015.
- Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming* 72(1), 2008.
- Bernd Bruegge and Peter Hibbard. Generalized Path Expressions: A High Level Debugging Mechanism. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Sym. on High-Level Debugging*, 1983.
- Margaret Burnett, Curtis Cook, Omkar Pendse, Gregg Rothermel, Jay Summet, and Chris Wallace. End-User Software Engineering with Assertions in the Spreadsheet Paradigm. In *Proc. Intl. Conf. on Software Engineering*, 2003.
- Stephen Chang, Alex Knauth, and Ben Greenman. Type Systems as Macros. In *Proc. ACM Sym. Principles of Programming Languages*, 2017.
- Philippe Charles, Robert M. Fuhrer, Stanley M. Sutton Jr., Evelyn Duesterwald, and Jurgen Vinju. Accelerating the Creation of Customized, Language-Specific IDEs in Eclipse. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, 2009.
- Andrei Chis, Tudor Girba, and Oscar Nierstrasz. The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers. In *Proc. Software Language Engineering*, 2014.
- Jonathan Corley, Brian P. Eddy, Eugene Syriani, and Jeff Gray. Efficient and Scalable Omniscient Debugging for Model Transformations. *Software Quality Journal* 25(1), 2017.
- Julien Deantoni. Modeling the Behavioral Semantics of Heterogeneous Languages and Their Coordination. In *Proc. Architecture Centric Virtual Integration*, 2016.
- Mireille Ducasse. Coca: An Automated Debugger for C. In *Proc. Intl. Conf. on Software Engineering*, 1999.
- Christopher J. Dutchyn. AspectScheme—Aspects in Higher-Order Languages. In *Proc. Workshop on Scheme and Functional Programming*, 2012.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic Abstraction in Scheme. *Lisp and Symbolic Computation* 5(4), 1993.

Peter Eades. A Heuristic for Graph Drawing. *Congressus Numerantium* 42, 1984.

Eclipse. <https://eclipse.org>, 2017.

Sven Efftinge and Markus Volter. oAW xText: A Framework for Textual DSLs. In *Proc. Wksp. on Modeling Sym. at Eclipse Summit*, 2006.

Marc Eisenstadt. My Hairiest Bug War Stories. *Communications of the ACM* 40(4), 1997.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In *Proc. 1st Summit on Advances in Programming Languages*, 2015.

Mattias Felleisen and Daniel P. Friedman. A Calculus for Assignments in Higher-Order Languages. In *Proc. ACM Sym. Principles of Programming Languages*, 1987.

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *Proc. ACM Intl. Conf. Functional Programming*, 2009.

Matthew Flatt, Ryan Culpepper, Robert Bruce Findler, and David Darais. Macros that Work Together: Compile-Time Bindings, Partial Expressions, and Definition Contexts. *J. Functional Programming* 22(2), 2012.

Spencer P. Florence, Burke Fetscher, Matthew Flatt, William H. Temps, Tina Kiguradze, Dennis P. West, Charlotte Niznik, Paul R. Yarnold, Robert Bruce Findler, and Steven M. Belknap. POP-PL: A Patient-Oriented Prescription Programming Language. In *Proc. Generative Programming and Component Engineering*, 2015.

Thomas M.J. Fruchterman and Edward M. Reingold. Graph Drawing by Force-Directed Placement. *Software—Practice & Experience* 21(11), 1991.

Michael Golan and David R. Hanson. DUEL-A Very High-Level Debugging Language. In *Proc. Usenix Annual Technical Conf.*, 1993.

Pedro Rangei Henriques, Maria Joao Varanda Pereira, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic Generation of Language-Based Tools Using the LISA System. *IEE Proceedings - Software* 152(2), 2005.

JetBrains. Meta Programming System. <https://www.jetbrains.com/mps/>, 2004.

Mark Scott Johnson. The Design of a High-Level, Language-Independent Symbolic Debugging System. In *Proc. ACM '77 Annual Conf.*, 1977.

Lennart C. L. Kats and Eelco Visser. The Spoofox Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, 2010.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proc. European Conf. Object-Oriented Programming*, 2001.

- Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Longtier, and John Irwin. Aspect-Oriented Programming. In *Proc. European Conf. Object-Oriented Programming*, 1997.
- Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic Macro Expansion. In *Proc. ACM Sym. on Lisp and Functional Programming*, 1986.
- Holger Krahn, Bernhard Rumpe, and Steven Volkel. MontiCore: Modular Development of Textual Domain Specific Languages. In *Proc. Intl. Conf. Objects, Models, Components, Patterns*, 2008.
- Bil Lewis. Debugging Backwards in Time. In *Proc. Intl. Wksp. on Automated Debugging*, 2003.
- Xiangqi Li and Matthew Flatt. Medic: Metaprogramming and Trace-Oriented Debugging. In *Proc. Wksp. on Future Programming*, 2015.
- Xiangqi Li and Matthew Flatt. Debugging with Domain-Specific Events via Macros. In *Proc. Software Language Engineering*, 2017.
- Ricky T. Lindeman, Lennart C. L. Kats, and Eelco Visser. Declaratively Defining Domain-Specific Language Debuggers. In *Proc. Generative Programming and Component Engineering*, 2011.
- Guillaume Marceau, Gregory H. Cooper, Jonathan P. Spiro, Shriram Krishnamurthi, and Steven P. Reiss. The Design and Implementation of a Dataflow Language for Scriptable Debugging. *Automated Software Engineering* 14(1), 2006.
- Sean McDirmid. Usable Live Programming. In *Proc. Sym. on New Ideas, New Paradigms, and Reflections on Programming & Software*, pp. 53–62, 2013.
- Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys* 37(4), 2005.
- Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, 1993.
- Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho. Dalek: A GNU, Improved Programmable Debugger. In *Proc. USENIX Technical Conf.*, 1990.
- Guillaume Pothier, Eric Tanter, and Jose Piquer. Scalable Omniscient Debugging. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, 2007.
- Olaf Spinczyk, Andreas Gal, and Wolfgang Schroder-Preikschat. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *Proc. Intl. Conf. on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, 2002.
- Richard Stallman, Roland Pesch, and Stan Shebs. Debugging with GDB. *Free Software Foundation*, 2002.
- Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as Libraries. In *Proc. ACM Conf. Programming Language Design and Implementation*, 2011.
- Yoshiyuki Usui and Shigeru Chiba. Bugdel: An Aspect-Oriented Debugging System. In *Proc. Asia-Pacific Software Engineering Conf.*, 2005.

- Mark G. J. Van Den Brand, B. Cornelissen, Pieter A. Olivier, and Jurgen J. Vinju. TIDE: A Generic Debugging Framework — Tool Demonstration. *Electronic Notes in Theoretical Computer Science* 141(4), 2005.
- Arie van Deursen, Jan Heering, and Paul Klint. Language Prototyping: An Algebraic Specification Approach. World Scientific Publishing Company, 1996.
- Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices* 35(6), 2000.
- David Van Horn and Matthew Might. Abstracting Abstract Machines. In *Proc. ACM Intl. Conf. Functional Programming*, 2011.
- Simon Van Mierlo, Claudio Gomes, and Hans Vangheluwe. Explicit Modelling and Synthesis of Debuggers for Hybrid Simulation Languages. In *Proc. Sym. on Theory of Modeling and Simulation*, 2017.
- Bret Victor. Learnable Programming. <http://worrydream.com/LearnableProgramming/>, 2012.
- Eelco Visser. Syntax Definition for Language Prototyping. PhD dissertation, University of Amsterdam, 1997.
- Martin P. Ward. Language Oriented Programming. *Software—Concepts and Tools* 15(4), 1994.
- Phil Winterbottom. Acid: A Debugger Built From A Language. In *Proc. USENIX Annual Technical Conf.* , 1994.
- Hui Wu, Jeff Gray, and Marjan Mernik. Grammar-Driven Generation of Domain-Specific Language Debuggers. *Software—Practice & Experience* 38(10), 2008.
- Hui Wu, Jeff Gray, Suman Roychoudhury, and Marjan Mernik. Weaving a Debugging Aspect into Domain-Specific Language Grammars. In *Proc. Sym. on Applied Computing*, 2005.
- Andreas Zeller. Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann Publishers, 2005.